Shell Lab

2018-13627 Kim Gideok

1. Implementation

1.1 eval

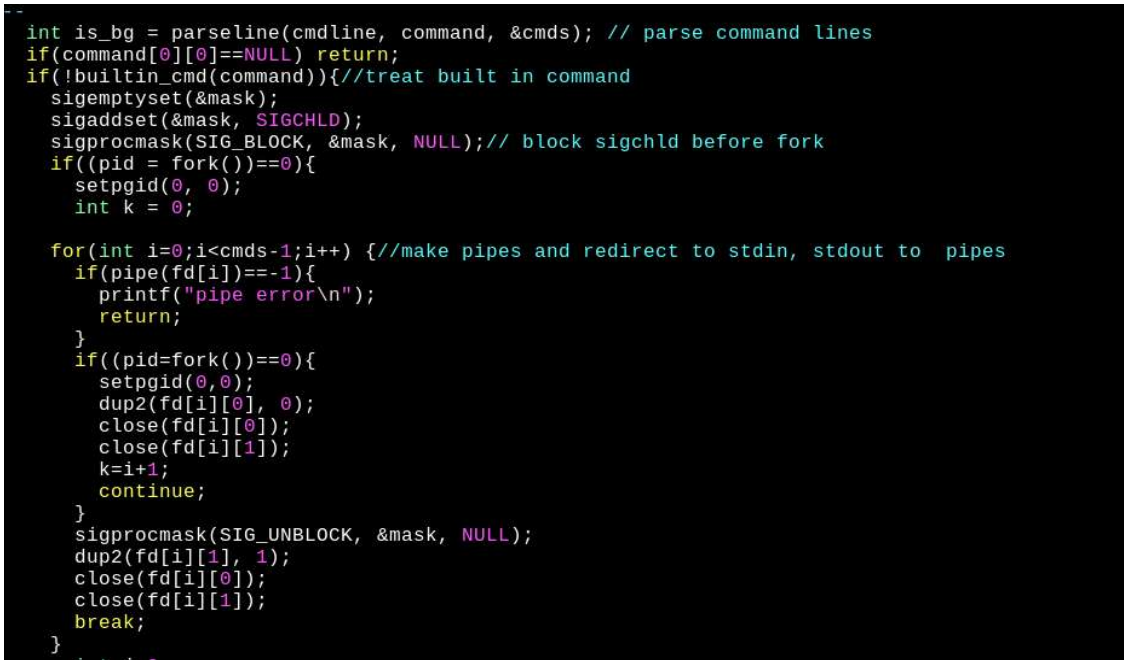
First, parse cmdline to make it to be treated easily.(using parseline)

cmdline -> command[MAXPIPES][MAXARGS]

Concurrently, parseline gives how many pipes we need.(to cmds)

Then, I checked if the command is built-in command by using builtin_cmd.

I blocked the SIGCHLD until it forks the child process.



```
  int is_bg = parseline(cmdline, command, &cmds); // parse command lines
  if(command[0][0]==NULL) return;
  if(!builtin_cmd(command)){//treat built in command
    sigemptyset(&mask);
    sigaddset(&mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &mask, NULL);// block sigchld before fork
    if((pid = fork())==0){
      setpgid(0, 0);
      int k = 0;

    for(int i=0;i<cmds-1;i++) {//make pipes and redirect to stdin, stdout to  pipes
      if(pipe(fd[i])==-1){
        printf("pipe error\n");
        return;
      }
      if((pid=fork())==0){
        setpgid(0,0);
        dup2(fd[i][0], 0);
        close(fd[i][0]);
        close(fd[i][1]);
        k=i+1;
        continue;
      }
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      dup2(fd[i][1], 1);
      close(fd[i][0]);
      close(fd[i][1]);
      break;
    }
```

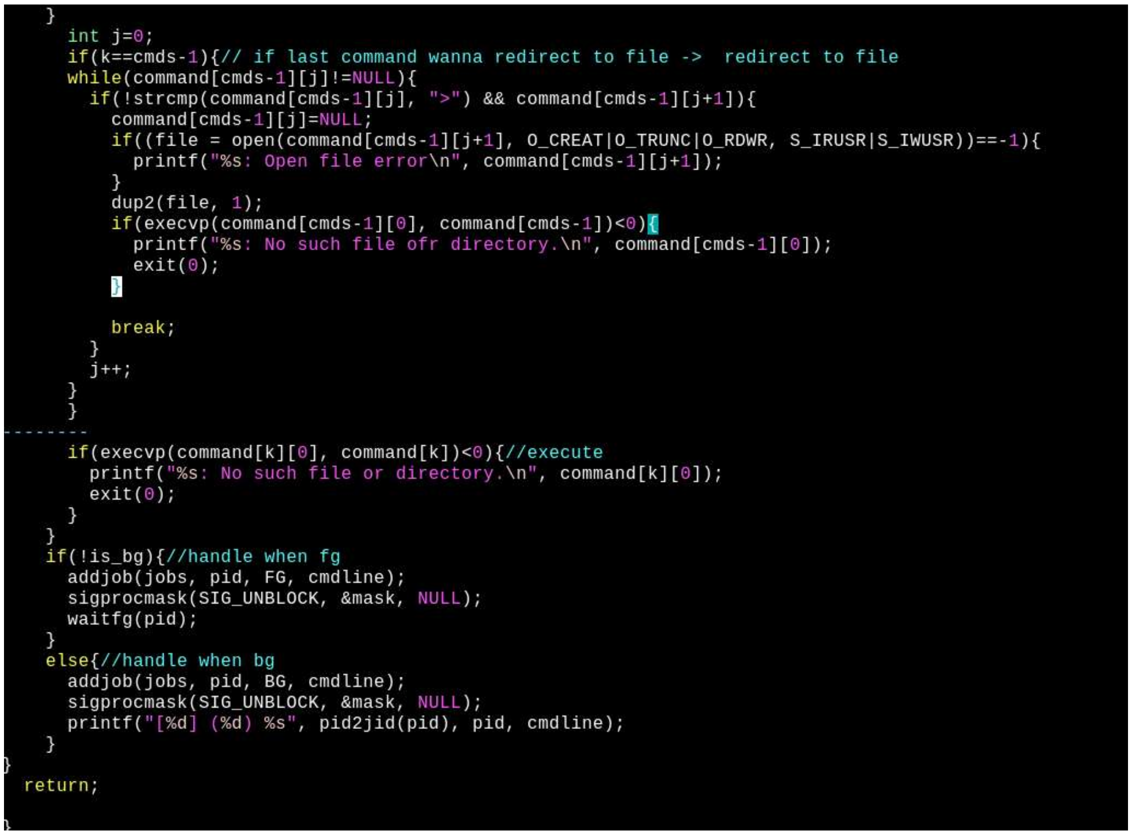
Next, I used pipe and fork to implement pipe operations. (ls | sort)

Also, I redirect stdin, stdout to pipes.

Then, each process run command + give and take input output from pipes.

If the last command use file redirection, I handled it using open file and redirection(dup2).

Then, the main process add job and wait for child process if the job is fg.



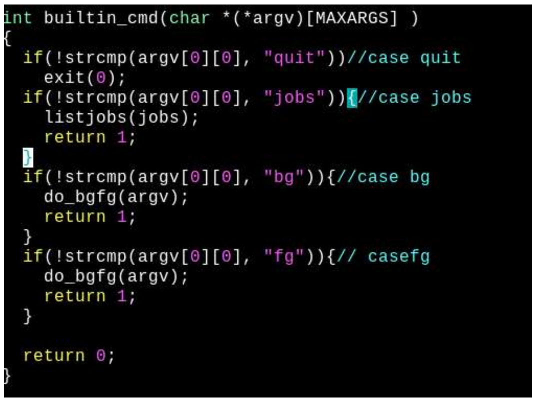
```
    }
      int j=0;
      if(k==cmds-1){// if last command wanna redirect to file ->  redirect to file
      while(command[cmds-1][j]!=NULL){
        if(!strcmp(command[cmds-1][j], ">") && command[cmds-1][j+1]){
          command[cmds-1][j]=NULL;
          if((file = open(command[cmds-1][j+1], O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR))==-1){
            printf("%s: Open file error\n", command[cmds-1][j+1]);
          }
          dup2(file, 1);
          if(execvp(command[cmds-1][0], command[cmds-1])<0){
            printf("%s: No such file ofr directory.\n", command[cmds-1][0]);
            exit(0);
          }

          break;
        }
        j++;
      }
      }
--------
      if(execvp(command[k][0], command[k])<0){//execute
        printf("%s: No such file or directory.\n", command[k][0]);
        exit(0);
      }
    }
    if(!is_bg){//handle when fg
      addjob(jobs, pid, FG, cmdline);
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      waitfg(pid);
    }
    else{//handle when bg
      addjob(jobs, pid, BG, cmdline);
      sigprocmask(SIG_UNBLOCK, &mask, NULL);
      printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
    }
}
  return;
}
```

1.2. builtin_cmd

Get the command and its first command is built-in command(quit, jobs, fg, bg)

Immediately runs it.



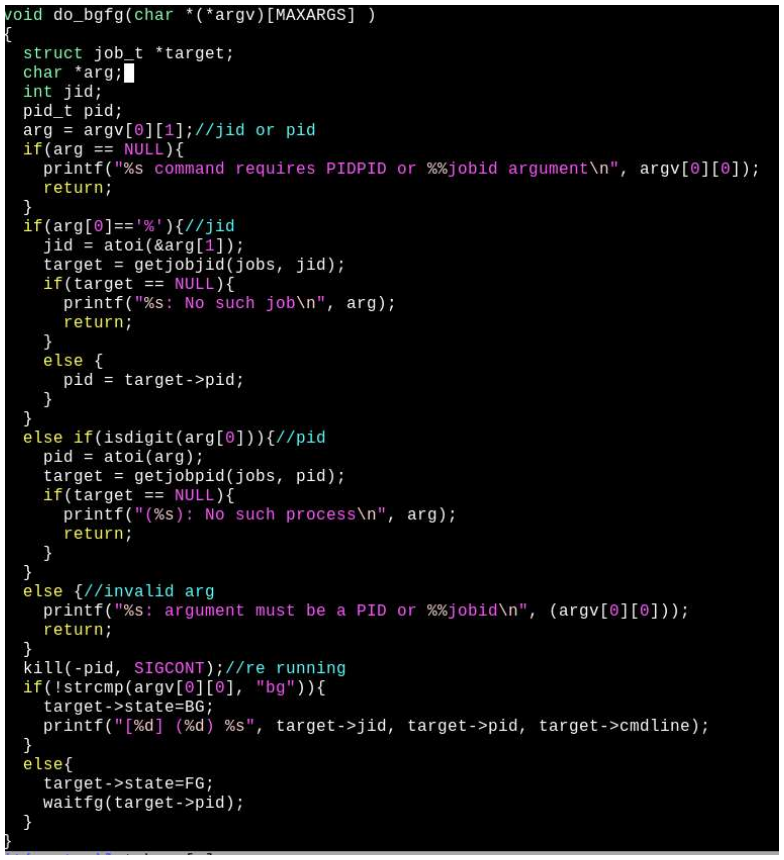
```
int builtin_cmd(char *(*argv)[MAXARGS] )
{
  if(!strcmp(argv[0][0], "quit"))//case quit
    exit(0);
  if(!strcmp(argv[0][0], "jobs")){//case jobs
    listjobs(jobs);
    return 1;
  }
  if(!strcmp(argv[0][0], "bg")){//case bg
    do_bgfg(argv);
    return 1;
  }
  if(!strcmp(argv[0][0], "fg")){// casefg
    do_bgfg(argv);
    return 1;
  }

  return 0;
}
```

1.3. do_bgfg

Find jid(with %) or pid.

If the job is valid, change it bg or fg and reruns it.

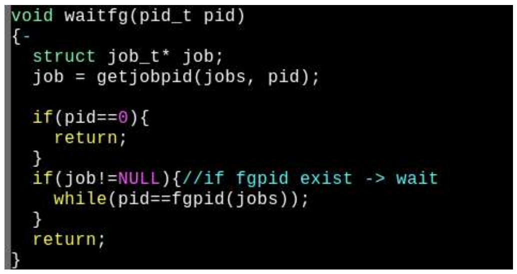


```
void do_bgfg(char *(*argv)[MAXARGS] )
{
  struct job_t *target;
  char *arg;
  int jid;
  pid_t pid;
  arg = argv[0][1];//jid or pid
  if(arg == NULL){
    printf("%s command requires PIDPID or %%jobid argument\n", argv[0][0]);
    return;
  }
  if(arg[0]=='%'){//jid
    jid = atoi(&arg[1]);
    target = getjobjid(jobs, jid);
    if(target == NULL){
      printf("%s: No such job\n", arg);
      return;
    }
    else {
      pid = target->pid;
    }
  }
  else if(isdigit(arg[0])){//pid
    pid = atoi(arg);
    target = getjobpid(jobs, pid);
    if(target == NULL){
      printf("(%s): No such process\n", arg);
      return;
    }
  }
  else {//invalid arg
    printf("%s: argument must be a PID or %%jobid\n", (argv[0][0]));
    return;
  }
  kill(-pid, SIGCONT);//re running
  if(!strcmp(argv[0][0], "bg")){
    target->state=BG;
    printf("[%d] (%d) %s", target->jid, target->pid, target->cmdline);
  }
  else{
    target->state=FG;
    waitfg(target->pid);
  }
}
```

1.4. waitfg

This function waits for foreground job.

Similar to waitpid.

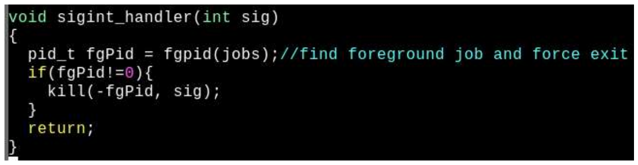


```
void waitfg(pid_t pid)
{
  struct job_t* job;
  job = getjobpid(jobs, pid);

  if(pid==0){
    return;
  }
  if(job!=NULL){//if fgpid exist -> wait
    while(pid==fgpid(jobs));
  }
  return;
}
```

1.5. sigint_handler

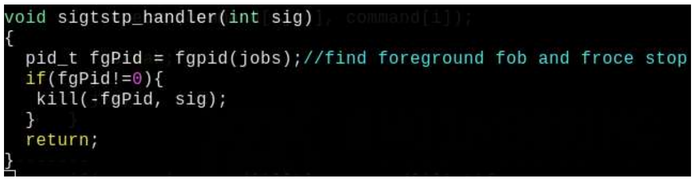
Find foreground jobs and kill them.



```
void sigint_handler(int sig)
{
  pid_t fgPid = fgpid(jobs);//find foreground job and force exit
  if(fgPid!=0){
    kill(-fgPid, sig);
  }
  return;
}
```

1.6. sigtstp_handler

Find foreground jobs and stop them.



```
void sigtstp_handler(int sig)
{
  pid_t fgPid = fgpid(jobs);//find foreground fob and froce stop
  if(fgPid!=0){
   kill(-fgPid, sig);
  }
  return;
}
```

1.7. sigchld_handler

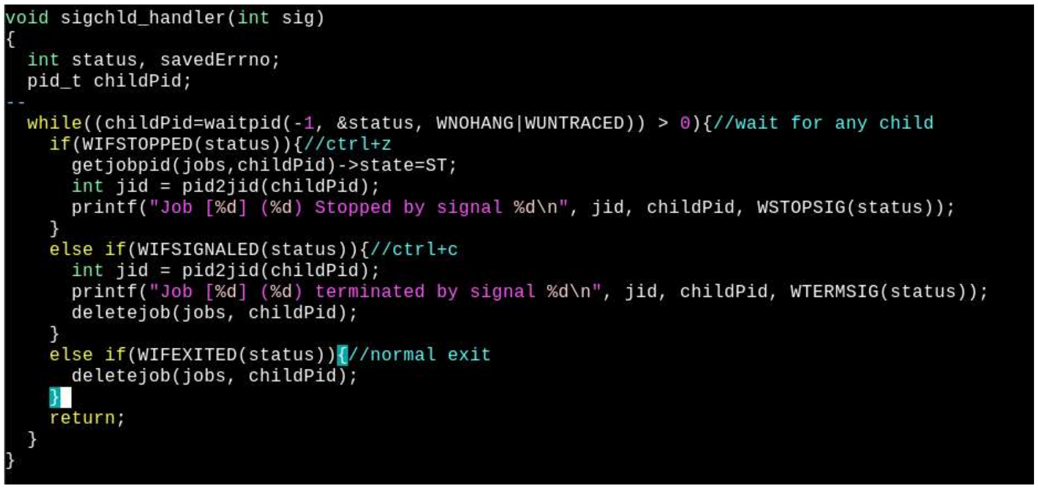
This handler checks children's status and do something according to the status.

Handler get child's status with waitpid function.

If the status is stopped by ctrl-z, changes the job state of the child. + message.

If the status is terminated by ctrl-c, delete the job. + message.

Just delete job when the child exit successfully.



```
void sigchld_handler(int sig)
{
  int status, savedErrno;
  pid_t childPid;
--
  while((childPid=waitpid(-1, &status, WNOHANG|WUNTRACED)) > 0){//wait for any child
    if(WIFSTOPPED(status)){//ctrl+z
      getjobpid(jobs,childPid)->state=ST;
      int jid = pid2jid(childPid);
      printf("Job [%d] (%d) Stopped by signal %d\n", jid, childPid, WSTOPSIG(status));
    }
    else if(WIFSIGNALED(status)){//ctrl+c
      int jid = pid2jid(childPid);
      printf("Job [%d] (%d) terminated by signal %d\n", jid, childPid, WTERMSIG(status));
      deletejob(jobs, childPid);
    }
    else if(WIFEXITED(status)){//normal exit
      deletejob(jobs, childPid);
    }
    return;
  }
}
```

2. Difficulties

2.1 setpgid

At first, I forgot to use setpgid after I forked child.

After that, everything didn't work well.

Even kill function didn't worked as my thoughts.

Quite long times later, I found I should user setpgid to set group id.

After that, everything worked well.

2.2 redirection

I had two worries.

First is that how to determine redirection tag (>)

Second is that how to handle command if the process find redirection tag.

I solved the first problem just turn around command and strcmp with '>'.

And then, I realized that execvp's second argument ends with null.

Therefore, I put Null instead of '>' and it worked well.

2.3 pipes

This was so difficult because I can't imagine what's happening after fork and pipe intuitionally.

So, I started with one pipe. Ex) ls | sort.

First, I thought that it is better to fork two children in main process.

But there was some troubles. First child process and second one cannot interactive directly.

They should talk each other passing main process.

So, I concluded that it was not that good idea.

Then, I came up with the idea that first child forks second child.

The code should be little more complicated when forking. But, the communication between processes got much easier and faster.

Then, I was also troubled with variables in many processes.

There were variables some of them should be shared, others should not be shared.

I broke through the problem reading the book and ppt again and again.

3. Conclusion

This lab was so challenged for me because I should treat fork.

Treating many process in one code is difficult for me.

But, I learned shared/unshared variables, using streak of pipes, input output redirections from this lab.

I was so pleased to remove some repulsion about fork.

But I wasn’t satisfied with using signal masks.(I think I don’t utilize them well) I want to learn more about when and where I should use signal masks.