Shell Lab

수리과학부 2019-16022 박채연

1. 실행결과

Redirection을 이용하여 test01부터 test15까지의 수행결과 및 rtest01부터 rtest15까지의 수행결과를 각각의 텍스트 파일에 저장하였다. 그리고 linux의 diff 명령어를 통해 두 텍스트 파일의 다른 부분을 출력하였다. 한편, diff 명령어의 경우 두 파일의 다른 부분을 줄 단위로 출력하기 때문에 해당 줄에서 어떤 부분이 다른지 알기가 어렵다. 따라서 WinMerge 프로그램을 이용하여 어떤 부분이 다른지 또한 체크하였다. 'tsh'와 'tshref'의 차이, 'test'와 'rtest'의 차이, pid값의 차이를 제외하고 모든 부분이 동일함을 확인할 수 있었다. 아래는 총 15개의 테스트 결과에 대해 WinMerge 프로그램 수행결과를 캡쳐한 결과이다. 왼쪽은 직접 구현한 쉘 프로그램의 출력 결과이고, 오른쪽은 정답 출력 결과이다.

(1) trace01.txt

```
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

```
./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
#
# trace01.txt - Properly terminate on EOF.
#
```

(2) trace02.txt

```
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

```
./sdriver.pl -t trace02.txt -s ./tshref -a "-p"
#
# trace02.txt - Process builtin quit command.
#
```

(3) trace03.txt

```
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
```

```
./sdriver.pl -t trace03.txt -s ./tshref -a "-p"
#
# trace03.txt - Run a foreground job.
#
tsh> quit
```

(4) trace04.txt

```
./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (3493) ./myspin 1 &
```

```
./sdriver.pl -t trace04.txt -s ./tshref -a "-p"
#
# trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (3637) ./myspin 1 &
```

(5) trace05.txt

```
./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (3499) ./myspin 2 &
tsh> ./myspin 3 &
[2] (3501) ./myspin 3 &
tsh> jobs
[1] (3499) Running ./myspin 2 &
[2] (3501) Running ./myspin 3 &
```

```
./sdriver.pl -t trace05.txt -s ./tshref -a "-p"
#
# trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (3643) ./myspin 2 &
tsh> ./myspin 3 &
[2] (3645) ./myspin 3 &
tsh> jobs
[1] (3643) Running ./myspin 2 &
[2] (3645) Running ./myspin 3 &
```

(6) trace06.txt

```
./sdriver.pl -t trace06.txt -s ./tsh -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (3508) terminated by signal 2
```

```
./sdriver.pl -t trace06.txt -s ./tshref -a "-p"
#
# trace06.txt - Forward SIGINT to foreground job.
#
tsh> ./myspin 4
Job [1] (3652) terminated by signal 2
```

(7) trace07.txt

```
./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (3514) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3516) terminated by signal 2
tsh> jobs
[1] (3514) Running ./myspin 4 &
```

```
./sdriver.pl -t trace07.txt -s ./tshref -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (3658) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3660) terminated by signal 2
tsh> jobs
[1] (3658) Running ./myspin 4 &
```

(8) trace08.txt

```
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (3523) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3525) stopped by signal 20
tsh> jobs
[1] (3523) Running ./myspin 4 &
[2] (3525) Stopped ./myspin 5
```

```
./sdriver.pl -t trace08.txt -s ./tshref -a "-p"
#
# trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (3667) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3669) stopped by signal 20
tsh> jobs
[1] (3667) Running ./myspin 4 &
[2] (3669) Stopped ./myspin 5
```

(9) trace09.txt

```
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (3532) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3534) stopped by signal 20
tsh> jobs
[1] (3532) Running ./myspin 4 &
[2] (3534) Stopped ./myspin 5
tsh> bg %2
[2] (3534) ./myspin 5
tsh> jobs
[1] (3532) Running ./myspin 4 &
[2] (3534) Running ./myspin 5
```

```
./sdriver.pl -t trace09.txt -s ./tshref -a "-p"
#
# trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (3676) ./myspin 4 &
tsh> ./myspin 5
Job [2] (3678) stopped by signal 20
tsh> jobs
[1] (3676) Running ./myspin 4 &
[2] (3678) Stopped ./myspin 5
tsh> bg %2
[2] (3678) ./myspin 5
tsh> jobs
[1] (3676) Running ./myspin 4 &
[2] (3678) Running ./myspin 5
```

(10) trace10.txt

```
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (3543) ./myspin 4 &
tsh> fg %1
Job [1] (3543) stopped by signal 20
tsh> jobs
[1] (3543) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

```
./sdriver.pl -t trace10.txt -s ./tshref -a "-p"
#
# trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (3687) ./myspin 4 &
tsh> fg %1
Job [1] (3687) stopped by signal 20
tsh> jobs
[1] (3687) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
```

(11) trace11.txt

```
./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (3553) terminated by signal 2
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/do
  165 pts/2    Ss     0:01 -bash
 3548 pts/2    S+     0:00 make test11
 3549 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tsh -a "-p"
 3550 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tsh -a -p
 3551 pts/2    S+     0:00 ./tsh -p
 3556 pts/2    R      0:00 /bin/ps a
```

```
./sdriver.pl -t trace11.txt -s ./tshref -a "-p"
#
# trace11.txt - Forward SIGINT to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (3697) terminated by signal 2
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/do
  165 pts/2    Ss     0:01 -bash
 3692 pts/2    S+     0:00 make rtest11
 3693 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace11.txt -s ./tshref -a "-p"
 3694 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace11.txt -s ./tshref -a -p
 3695 pts/2    S+     0:00 ./tshref -p
 3700 pts/2    R      0:00 /bin/ps a
```

(12) trace12.txt

```
./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
#
# trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (3562) stopped by signal 20
tsh> jobs
[1] (3562) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/do
  165 pts/2    Ss     0:01 -bash
 3557 pts/2    S+     0:00 make test12
 3558 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
 3559 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tsh -a -p
 3560 pts/2    S+     0:00 ./tsh -p
 3562 pts/2    T      0:00 ./mysplit 4
 3563 pts/2    T      0:00 ./mysplit 4
 3566 pts/2    R      0:00 /bin/ps a
```

```
./sdriver.pl -t trace12.txt -s ./tshref -a "-p"
#
# trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (3706) stopped by signal 20
tsh> jobs
[1] (3706) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/do
  165 pts/2    Ss     0:01 -bash
 3701 pts/2    S+     0:00 make rtest12
 3702 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace12.txt -s ./tshref -a "-p"
 3703 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace12.txt -s ./tshref -a -p
 3704 pts/2    S+     0:00 ./tshref -p
 3706 pts/2    T      0:00 ./mysplit 4
 3707 pts/2    T      0:00 ./mysplit 4
 3710 pts/2    R      0:00 /bin/ps a
```

(13) trace13.txt

```
./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (3572) stopped by signal 20
tsh> jobs
[1] (3572) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/do
  165 pts/2    Ss     0:01 -bash
 3567 pts/2    S+     0:00 make test13
 3568 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
 3569 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
 3570 pts/2    S+     0:00 ./tsh -p
 3572 pts/2    T      0:00 ./mysplit 4
 3573 pts/2    T      0:00 ./mysplit 4
 3576 pts/2    R      0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/do
  165 pts/2    Ss     0:01 -bash
 3567 pts/2    S+     0:00 make test13
 3568 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tsh -a "-p"
 3569 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tsh -a -p
 3570 pts/2    S+     0:00 ./tsh -p
 3579 pts/2    R      0:00 /bin/ps a
```

```
./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
#
# trace13.txt - Restart every stopped process in process group
#
tsh> ./mysplit 4
Job [1] (3716) stopped by signal 20
tsh> jobs
[1] (3716) Stopped ./mysplit 4
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --d
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/doc
  165 pts/2    Ss     0:01 -bash
 3711 pts/2    S+     0:00 make rtest13
 3712 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
 3713 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshref -a -p
 3714 pts/2    S+     0:00 ./tshref -p
 3716 pts/2    T      0:00 ./mysplit 4
 3717 pts/2    T      0:00 ./mysplit 4
 3720 pts/2    R      0:00 /bin/ps a
tsh> fg %1
tsh> /bin/ps a
  PID TTY      STAT   TIME COMMAND
  133 pts/0    Ssl+   0:02 /mnt/wsl/docker-desktop/docker-desktop-user-distro proxy --d
  149 pts/1    Ssl+   0:08 docker serve --address unix:///home/chaeyeon/.docker/run/doc
  165 pts/2    Ss     0:01 -bash
 3711 pts/2    S+     0:00 make rtest13
 3712 pts/2    S+     0:00 /bin/sh -c ./sdriver.pl -t trace13.txt -s ./tshref -a "-p"
 3713 pts/2    S+     0:00 /usr/bin/perl ./sdriver.pl -t trace13.txt -s ./tshref -a -p
 3714 pts/2    S+     0:00 ./tshref -p
 3723 pts/2    R      0:00 /bin/ps a
```

(14) trace14.txt

```
./sdriver.pl -t trace14.txt -s ./tsh -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (3587) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
Job [1] (3587) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (3587) ./myspin 4 &
tsh> jobs
[1] (3587) Running ./myspin 4 &
```

```
./sdriver.pl -t trace14.txt -s ./tshref -a "-p"
#
# trace14.txt - Simple error handling
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 4 &
[1] (3731) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: argument must be a PID or %jobid
tsh> bg a
bg: argument must be a PID or %jobid
tsh> fg 9999999
(9999999): No such process
tsh> bg 9999999
(9999999): No such process
tsh> fg %2
%2: No such job
tsh> fg %1
Job [1] (3731) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (3731) ./myspin 4 &
tsh> jobs
[1] (3731) Running ./myspin 4 &
```

(15) trace15.txt

```
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (3606) terminated by signal 2
tsh> ./myspin 3 &
[1] (3608) ./myspin 3 &
tsh> ./myspin 4 &
[2] (3610) ./myspin 4 &
tsh> jobs
[1] (3608) Running ./myspin 3 &
[2] (3610) Running ./myspin 4 &
tsh> fg %1
Job [1] (3608) stopped by signal 20
tsh> jobs
[1] (3608) Stopped ./myspin 3 &
[2] (3610) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (3608) ./myspin 3 &
tsh> jobs
[1] (3608) Running ./myspin 3 &
[2] (3610) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

```
./sdriver.pl -t trace15.txt -s ./tshref -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found
tsh> ./myspin 10
Job [1] (3750) terminated by signal 2
tsh> ./myspin 3 &
[1] (3752) ./myspin 3 &
tsh> ./myspin 4 &
[2] (3754) ./myspin 4 &
tsh> jobs
[1] (3752) Running ./myspin 3 &
[2] (3754) Running ./myspin 4 &
tsh> fg %1
Job [1] (3752) stopped by signal 20
tsh> jobs
[1] (3752) Stopped ./myspin 3 &
[2] (3754) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (3752) ./myspin 3 &
tsh> jobs
[1] (3752) Running ./myspin 3 &
[2] (3754) Running ./myspin 4 &
tsh> fg %1
tsh> quit
```

2. 어떻게 구현했는지

(1) 시스템콜 래퍼 함수들

```c
void unix_error(char *msg)
{
    fprintf(stdout, "%s: %s\n", msg, strerror(errno));
    exit(1);
}
```

위와 같이 문자열이 주어졌을 때 오류메시지를 출력하고 exit하는 unix_error 함수를 래퍼함수 내에서 사용하였다.

```c
void Sleep(int num){
    int n;
    if ((n=sleep(num))<0)
        unix_error("sleep error occured");
}
```

Sleep은 sleep 시스템콜 함수에 대한 래퍼함수이다. 에러 발생시 sleep 함수는 음수값을 리턴하는데 이때 unix_error 함수를 호출하여 에러메시지를 출력하고 exit하였다.

```c
void Write(int fd, char *str, int n){
    int nwrite=0;
    if ((nwrite=write(1, str, n)) < 0)
        unix_error("write error occured");
}
```

Write은 write 시스템콜 함수에 대한 래퍼함수이다. 에러 발생시 write 함수는 음수값을 리턴하는데 이때 unix_error 함수를 호출하여 에러메시지를 출력하고 exit하였다.

```
void Kill(pid_t pid, int sig){
    if (kill(pid, sig))
        unix_error("kill error occured.\n");
}
```

Kill함수는 kill 시스템콜 함수에 대한 래퍼함수이다. kill 함수는 에러 발생시 -1을, 성공시 0을 리턴한다. 에러 발생시 unix_error 함수를 호출하여 에러메시지를 출력하고 exit하였다.

```
pid_t Fork(void){
    pid_t pid;
    if ((pid=fork())<0)
        unix_error("fork error occured.\n");
    return pid;
}
```

Fork 함수는 fork 시스템콜 함수에 대한 래퍼함수이다. fork 함수는 에러 발생시 음수값을, 성공시 자식 프로세스의 pid를 리턴한다. 에러 발생시 unix_error 함수를 호출하여 에러메시지를 출력하고 exit 하였다. 에러가 발생하지 않은 경우 자식 프로세스의 pid를 리턴하였다.

```
void Sigemptyset(sigset_t *sigvec){
    if (sigemptyset(sigvec))
        unix_error("sigemptyset error occured");
}
```

Sigemptyset은 sigemtyset 시그널 함수에 대한 래퍼함수이다. sigemptyset 함수는 에러 발생시 -1을, 성공시 0을 리턴한다. 에러 발생시 unix_error 함수를 호출하여 에러메시지를 출력하고 exit하였다.

```
void Sigaddset(sigset_t *sigvec, int sig){
    if (sigaddset(sigvec, sig))
        unix_error("sigaddset error occured");
}
```

Sigaddset은 sigaddset 시그널 함수에 대한 래퍼함수이다. sigaddset 함수는 에러 발생시 -1을, 성공시 0을 리턴한다. 에러 발생시 unix_error 함수를 호출하여 에러메시지를 출력하고 exit하였다.

```
void Sigprocmask(int sig, sigset_t *new_sigvec, sigset_t *old_sigvec){
    if (sigprocmask(sig, new_sigvec, old_sigvec))
        unix_error("sigprocmask error occured");
}
```

Sigprocmask는 sigprocmask 시스템콜 함수에 대한 래퍼함수이다. sigprocmask 함수는 에러 발생시 -1을, 성공시 0을 리턴한다. 에러 발생시 unix_error 함수를 호출하여 에러메시지를 호출하고 exit하였다.

```
void Setpgid(pid_t pid, pid_t pgid){
    if (setpgid(pid, pgid))
        unix_error("setpgid error occured");
}
```

Setpgid는 setpgid 시스템콜 함수에 대한 래퍼함수이다. setpgid 함수는 에러 발생시 -1을, 성공시

0을 리턴한다. 에러 발생시 unix_error 함수를 호출하여 에러메시지를 출력하고 exit하였다.

(2) inside_handler_strlen, inside_handler_itoa, inside_handler_printf

inside_handler_strlen, inside_handler_itoa와 inside_handler_printf는 sigchld_handler 내부에서 사용하기 위해 구현한 helper 함수이다.

```
int inside_handler_strlen(char *str){
    int idx=0;

    while(str[idx])
        idx++;
    return idx;
}
```

```
void inside_handler_itoa(int num, char *str){
    int idx = 0, l, r;

    //Change each digit into char type and store them at the str argument
    while(num!=0){
        str[idx]=(num%10)+'0';
        idx+=1;
        num/=10;
    }
    str[idx]=0;

    //Reverse the str array
    l=0;
    r=idx-1;
    while (l<r){
        char c=str[l];
        str[l]=str[r];
        str[r]=c;
        l+=1;
        r-=1;
    }
    return ;
}
```

inside_handler_strlen는 주어진 문자열의 길이를 리턴하는 함수이고, inside_handler_itoa는 숫자를 문자열의 형태로 변환해주는 함수이다.

```
/* inside_handler_printf - async-signal-IO functions for signal handlers */
void inside_handler_printf(int jid, int pid, char *terminated_or_stopped, int status){
    char *str1="Job [";
    char *str2="] (";
    char *str3=") ";
    char *str4=" by signal ";
    char *str5="\n";
    char jstr[100], pstr[100], sstr[100];
    inside_handler_itoa(jid, jstr);
    inside_handler_itoa(pid, pstr);
    inside_handler_itoa(status, sstr);

    Write(1, str1, inside_handler_strlen(str1));
    Write(1, jstr, inside_handler_strlen(jstr));
    Write(1, str2, inside_handler_strlen(str2));
    Write(1, pstr, inside_handler_strlen(pstr));
    Write(1, str3, inside_handler_strlen(str3));
    Write(1, terminated_or_stopped, inside_handler_strlen(terminated_or_stopped));
    Write(1, str4, inside_handler_strlen(str4));
    Write(1, sstr, inside_handler_strlen(sstr));
    Write(1, str5, 1);
}
```

또한, inside_handler_printf 는 시그널 핸들러 내부에서 사용할 수 있는 async-signal-safe 함수로써, 만약 jid가 2, pid가 2781인 job에 SIGSTP signal이 전달되면 "Job [2] (2781) stopped by signal 20"와 같은 문자열을 터미널에 출력하는 역할을 한다. printf와 같은 Standard IO 함수의 경우 async-signal-safe 하지 않기 때문에 시그널 핸들러 내부에서 사용해서는 안된다. 따라서 async-signal-safe 함수인 write의 래퍼함수를 이용하여 inside_handler_printf를 구현하였다.

(3) eval

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];
    int bg;
    pid_t pid;
    sigset_t sigvec;

    Sigemptyset(&sigvec);

    //Parse cmdline
    bg = parseline(cmdline, argv);

    //Return if there is no argument
    if (argv[0]==NULL)
        return;
```

이는 eval 함수의 전반부이다. 가장 먼저 cmdline을 parseline 함수를 이용하여 parse하고 적어도 하나의 argument가 있는지 체크하였다. 만약 어떤 argument도 주어지지 않았다면 리턴하였다.

```
//Return if there is no argument
if (argv[0]==NULL)
    return;

//If an argument is not a built-in command, then we need to call fork()
if (!builtin_cmd(argv)){
    //Block SIGCHLD signal
    Sigaddset(&sigvec, SIGCHLD);
    Sigprocmask(SIG_BLOCK, &sigvec, NULL);

    //Child process
    if (pid=Fork()){
        //Unblock the signal and set new process group id.
        Sigprocmask(SIG_UNBLOCK, &sigvec, NULL);
        Setpgid(0, 0);

        //Load and run new program
        if (execve(argv[0], argv, environ)<0){
            printf("%s: Command not found\n", argv[0]);
            exit(0);
        }
    }
    //Parent process
    else{
        //If it is a foreground job, then add the job, unblock signal, and wait for the froeground job to terminate.
        if (!bg){
            addjob(jobs, pid, FG, cmdline);
            Sigprocmask(SIG_UNBLOCK, &sigvec, NULL);
            waitfg(pid);
        }
        //If it is a background job, then add the job, unblock the signal, and print log message
        else{
            addjob(jobs, pid, BG, cmdline);
            Sigprocmask(SIG_UNBLOCK, &sigvec, NULL);
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        }
    }
}
return;
```

다음으로 주어진 명령어가 builtin 명령어인지 여부를 체크한다. 만약 builtin 명령어라면 즉시 실행하지만, builtin 명령어가 아니라면 fork()를 통해 자식 프로세스에서 해당 명령어를 수행하게 된다. 주어진 명령어가 builtin 명령어가 아닌 경우 SIGCHLD 시그널을 block 하고 자식 프로세스를 생성하였다.

자식 프로세스 내에서는 우선 block 했던 signal을 unblock 하고 Setpgid(0, 0)을 통해 해당 프로세스의 그룹 아이디를 해당 프로세스의 pid와 동일한 값으로 설정한다. 다음 execve 함수를 통해 새로운 프로그램을 로드 하고 실행하였다.

부모 프로세스 내에서는 우선 addjob을 수행하고나서 block 했던 signal을 unblock하였다. 만약 주어진 명령어가 'bg'라면 주어진 job의 jid, pid, cmdline을 포함한 로그 메시지를 출력하였다.

(4) builtin_cmd

```
int builtin_cmd(char **argv)
{
    //'quit', 'fg', 'bg', and 'jobs' are builtin commands

    //'quit' command terminates the shell
    if (!strcmp(argv[0], "quit")){
        exit(0);
    }
    //'fg' and 'bg' commands call do_bgfg functions
    else if (!strcmp(argv[0], "fg") || !strcmp(argv[0], "bg")){
        do_bgfg(argv);
        return 1;
    }
    //'jobs' command lists all background jobs
    else if (!strcmp(argv[0], "jobs")){
        listjobs(jobs);
        return 1;
    }

    return 0;     /* not a builtin command */
}
```

tsh의 built-in command는 'quit', 'jobs', 'bg', 'fg'로 총 4개이다. quit는 쉘을 종료하는 명령어이고, jobs은 모든 background에서 실행중인 jobs를 출력하는 명령어이다. 명령어가 bg 혹은 fg일 경우 do_bgfg(argv)를 호출하여 관련 작업을 수행할 수 있도록 하였다.

(5) do_bgfg

```
void do_bgfg(char **argv)
{
    int jid, pid, bg;
    struct job_t *job = NULL;

    //Check whether PID or JID was given as an argument
    if (!argv[1]){
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
```

```
//Case 1: JID
if (argv[1][0]=='%'){
    jid = atoi(argv[1]+1);

    //Even though argv[1] starts with '%', it is not JID.
    if (!jid){
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    else{
        job = getjobjid(jobs, jid);
        //If there is no such job with the given jid
        if (!job){
            printf("%s: No such job\n", argv[1]);
            return;
        }
    }
}
//Case 2: PID
else {
    pid = atoi(argv[1]);

    //It is not PID
    if (!pid){
        printf("%s: argument must be a PID or %%jobid\n", argv[0]);
        return;
    }
    else{
        job = getjobpid(jobs, pid);
        //If there is no such process with the given pid
        if (!job){
            printf("(%s): No such process\n", argv[1]);
            return ;
        }
    }
}
```

이는 do_bgfg 함수의 전반부이다. bg <job> 명령어는 <job> 에 SIGCONT 시그널을 보내 background에서 재실행될 수 있도록 하는 명령어이고, fg <job> 명령어는 <job>에 SIGCONT 시그널을 보내 foreground에서 재실행될 수 있도록 하는 명령어이다. 이때, <job>으로 jid 혹은 pid가 주어지는데, jid의 경우 %로 시작한다. 만약 jid 혹은 pid가 주어지지 않았거나 주어진 jid 혹은 pid가 jobs 내에 없는 값이라면 관련 에러 문구를 출력하고 리턴하였다.

```
    bg = !strcmp(argv[0], "bg");

    //If the given command is 'bg'
    if (bg){
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
        Kill(-(job->pid), SIGCONT);
    }
    //If the given command is 'fg'
    else{
        job->state = FG;
        Kill(-(job->pid), SIGCONT);
        waitfg(job->pid);
    }

    return;
}
```

만약 주어진 명령어가 'bg' 라면 주어진 jid 혹은 pid를 가지는 job의 state를 BG로 바꾸고, 해당 프로세스가 속하는 그룹의 모든 프로세스에 SIGCONT 시그널을 보내 restart할 수 있도록 하였다. 또한, printf() 함수를 통해 해당 job의 jid, pid와 cmdline을 출력하였다.

반면, 만약 주어진 명령어가 'fg' 라면 주어진 jid 혹은 pid를 가지는 job의 state를 FG로 바꾸고, 해당 프로세스가 속하는 그룹의 모든 프로세스에 SIGCONT 시그널을 보내 restart 할 수 있도록 하였다. 마지막으로 waitfg 함수를 호출하여 foreground job이 끝날 때까지 기다리도록 하였다.

(6) waitfg

```
void waitfg(pid_t pid)
{
    while (pid == fgpid(jobs))
        Sleep(1);
    return;
}
```

waitfg는 sleep 함수를 호출하여 foreground job이 끝날 때까지 기다리도록 하는 함수이다.

(7) sigchld_handler, sigint_handler, sigstp_handler

```
void sigchld_handler(int sig)
{
    int status;
    pid_t pid;

    //There is just a single bit indicating whether there is pending SIGCHLD signal or not.
    //Since the signals are not queued, we need to call a waitpid function with WNOHANG option in 'while'
    while ((pid=waitpid(-1, &status, WNOHANG | WUNTRACED))>0){
        //If the process terminated normally, then just delete the job with 'pid' from jobs
        if (WIFEXITED(status))
            deletejob(jobs, pid);
        //If the process terminated normally, then just delete the job with 'pid' from jobs
        else if (WIFSIGNALED(status)){
            inside_handler_printf(pid2jid(pid), pid, "terminated", WTERMSIG(status));
            deletejob(jobs, pid);
        }
        //If the process stopped, then print the state and change the state of the job to 'ST'
        else if (WIFSTOPPED(status)){
            inside_handler_printf(pid2jid(pid), pid, "stopped", WSTOPSIG(status));
            getjobpid(jobs, pid)->state=ST;
        }
    }
    // Handle waitpid error
    if (pid<0 && errno!=ECHILD){
        unix_error("waitpid error occured");
    }

    return;
}
```

sigchld_handler 함수는 SIGCHLD 시그널을 핸들링하는 함수이다. 커널은 각 signal type 마다 하나의 비트를 이용해 해당 타입의 시그널이 pending 상태인지 여부를 확인하는데, 만약 어떤 프로세스에서 SIGCHLD 시그널이 pending 상태라면 해당 프로세스에 추가로 SIGCHLD 시그널이 전달되었을 때 더 이상 Queue 되지 않고 버려진다. 따라서 while 문에서 waitpid(-1, &status, WNOHANG | WUNTRACED)의 반환 값을 체크하도록 코드를 작성하였다.

만약 waitpid가 0을 반환하면 이는 더 이상 정지 혹은 종료 상태인 자식 프로세스가 없다는 의

미이므로 while 문을 빠져나오게 된다. 만약 반환값이 양수라면, 이는 정지 혹은 종료 상태에 있는 자식 프로세스의 pid값이다. 만약 해당 프로세스가 정상적으로 종료되었다면 jobs에서 해당 job을 삭제하였다. 만약 해당 프로세스가 시그널에 의해 비정상적으로 종료되었거나 정지 상태에 놓여있다면 앞서 구현한 async-signal-safe 함수인 inside_handler_printf 함수를 이용해 해당 job의 pid, jid, 상태 등을 출력 형식에 맞추어 출력하였다. 또한, 시그널에 의해 종료된 프로세스의 경우에는 jobs에서 해당 job을 삭제하였고, 시그널에 의해 정지 상태에 놓인 프로세스의 경우에는 jobs 내에서 state를 ST로 바꾸어주었다.

```
void sigint_handler(int sig)
{
    //Get the pid of the foreground job
    pid_t pid = fgpid(jobs);

    //If there is a foreground job, terminate each process in the foreground group
    if (pid)
        Kill(-pid, sig);
    return;
}
```

sigint_handler 함수는 SIGINT 시그널이 전달되었을 때 이를 핸들링하는 함수이다. foreground에서 실행 중인 프로세스가 있다면, foreground process group에 속한 모든 프로세스를 종료시킨다. 만약 foreground에서 수행 중인 프로세스가 없다면 어떠한 변화도 발생하지 않는다.

```
void sigtstp_handler(int sig)
{
    //Get the pid of the foreground job
    pid_t pid = fgpid(jobs);

    //If there is a foreground job, stop each process in the foreground group
    if (pid)
        Kill(-pid, sig);
    return;
}
```

sigstp_handler 함수는 SIGSTP 시그널이 전달되었을 때 이를 핸들링하는 함수이다. foreground에서 실행 중인 프로세스가 있다면, foreground process group에 속한 모든 프로세스를 정지시킨다. 만약 foreground에서 수행 중인 프로세스가 없다면 어떠한 변화도 발생하지 않는다.

3. 어려웠던 점

본 과제를 구현할 때 모든 시스템콜 함수에 대해 오류 발생 여부를 확인하기 위해 반환값을 체크하는 코드를 포함해야했지만 시스템콜 함수를 호출할 때마다 해당 코드를 추가하니 가독성이 떨어지고 코드의 크기가 늘어나서 코드 정리에 어려움을 겪었다. 해결 방법에 대해 고민하던 중 교과서인 CSAPP에서는 래퍼함수를 따로 작성하여 반환값을 래퍼함수 내부에서 체크하여 오류 여부를 확인하는 기능을 유지하면서 가독성이 높은 프로그램을 작성했다는 것을 알게 되었다. 교과서와 같이 자주 호출되는 시스템 콜 함수에 대해서 따로 래퍼함수를 작성하여 반환 값 체크를 통

해 에러 발생 여부를 확인하도록 함으로써 코드 정리의 어려움을 해결할 수 있었다.

또한, 교수님께서 수업 시간에 프로세스가 signal을 receive하여 이를 handling할 때 우선 해당 signal을 block하고 handling이 끝나면 다시 unblock한다고 하셨었지만 수업자료에서 제공된 eval 함수에서는 이 과정이 모두 생략되어 있었다. 반면 본 과제에서는 해당 부분까지 모두 구현해야 했을 뿐만 아니라 sigemptyset, sigaddset, sigprocmask 등의 함수도 너무나 생소해서 어렵다고 느껴졌다. 다행히 랩 ppt 5페이지에 eval 함수 내의 로직이 대략적으로 제시되어 있었고 이를 따라 올바른 eval 함수를 구현할 수 있었다. 뿐만 아니라 어떤 순서로 쉘이 작동하는지 이해할 수 있는 좋은 기회였다.

4. 새롭게 배운 점

read나 write, fork 등 시스템콜 함수를 프로그램 내부에서 사용할 때마다 반환 값 체크를 통해 에러가 발생했는지 여부를 체크하고 에러 발생 시 exit하는 등의 처리가 필요함을 배웠다. 또한, 시스템콜 함수를 호출할 때마다 반환 값을 체크하는 코드를 추가하면 프로그램 전체 코드에 대한 가독성이 떨어질 수 있는데, 래퍼함수를 따로 작성하여 사용하면 가독성을 떨어뜨리지 않고 오류 여부를 확인하는 코드를 작성할 수 있다는 것을 알게 되었다.

또한, 본 과목을 수강하기 전에는 쉘이 어떻게 동작하는지 생각해본 적이 없었는데 본 과목을 수강하면서 쉘의 동작 과정을 깊이 이해할 수 있었다. 쉘의 execution은 read 및 evaluate의 연속이며, 백그라운드 명령어인지, 내장 명령어인지 여부에 따라 다르게 동작함을 eval 함수 구현을 통해 배울 수 있었다.

겨울방학동안 인턴을 하면서 파라미터를 바꾸어가며 쉘스크립트를 실행시키는 작업을 한 적이 있는데 쉘 스크립트 실행시간이 한 번에 약 20분이 넘었었다. 터미널을 계속해서 확인하다보니 집중이 깨지고 효율이 떨어져 여러 개의 쉘 스크립트를 한꺼번에 실행시키고 싶었는데, 당시 쉘에 대한 이해 부족으로 터미널을 여러 개 켜고 각각의 터미널에 한개씩 쉘 스크립트를 실행시켰었다. 그러나 본 과제를 수행하면서 프로세스 실행은 background 로도 이루어질 수 있음을 배웠다. 명령어의 끝에 '&'를 붙이면 해당 명령어가 background로 수행된다는 사실을 알 수 있었다. 만약 다음 번에 시간이 오래 걸리는 작업을 한꺼번에 실행하고 싶다면 터미널을 여러 개 켜는 것이 아니라 여러 개의 프로세스를 background로 실행시키는 방법을 적용할 수 있을 것 같다.