# 시스템 프로그래밍

Shell Lab1

2016.11.15

**황슬아**

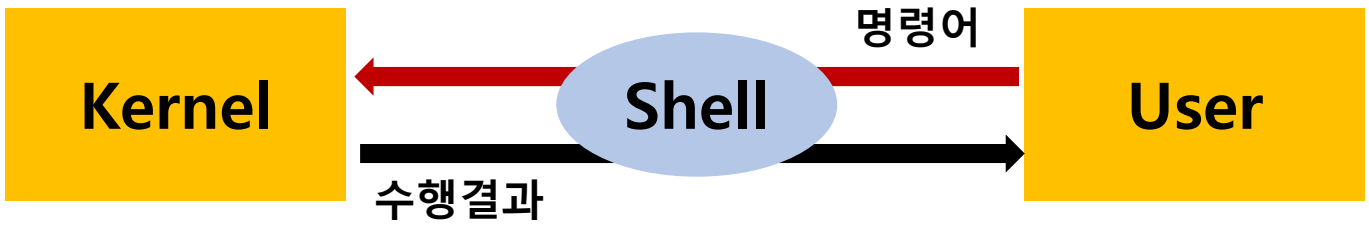
seula.hwang@cnu.ac.kr

# 개요

1. 실습명
   - ✓ Shell Lab
2. 목표
   - ✓ 작업 관리를 지원하는 간단한 Unix Shell 프로그램 구현과 이를 통한 프로세스의 제어와 시그널링(Signalling)의 개념 이해
3. 과제 진행
   - ✓ 수업시간에 배운 유닉스 지식을 활용하여 구현한다.
4. 구현사항
   - ✓ 기본적인 유닉스 쉘의 구현
   - ✓ 작업 관리(foreground / background)기능 및 쉘 명령어 구현

# Shell 이란?

1. 쉘(Shell)은 사용자와 Kernel을 연결시켜주는 인터페이스 역할을 한다.

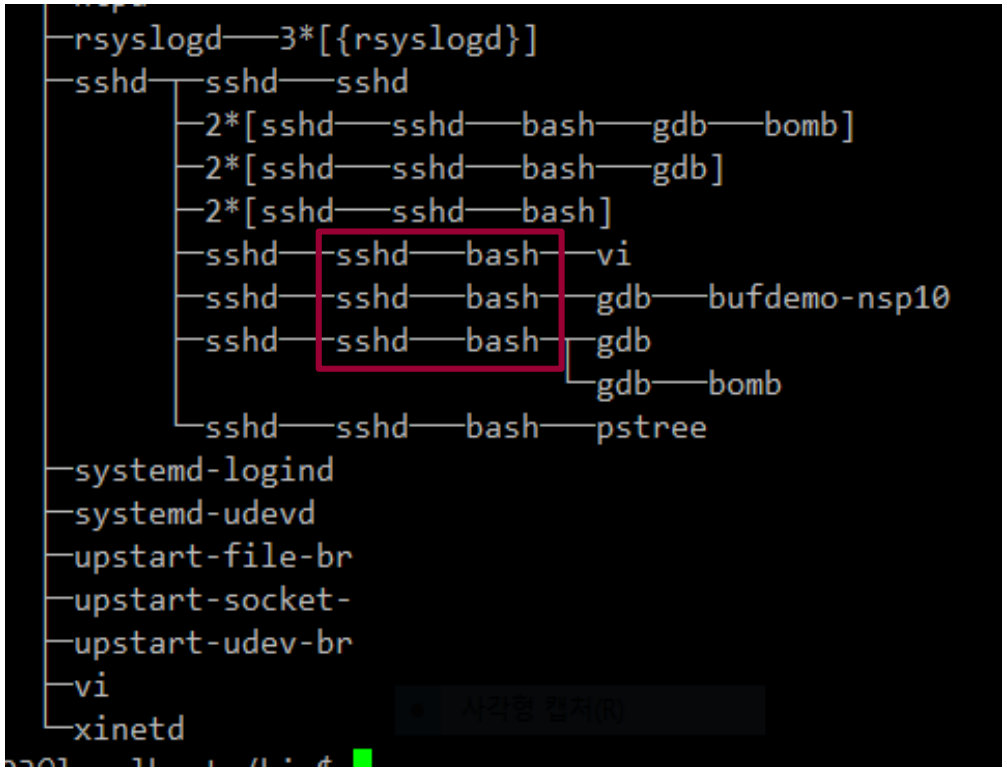



2. Shell의 기능

   1) 명령어 해석기 기능
      - 사용자가 입력한 명령어를 해석하고 커널에 전달하는 역할
      - “ls” 명령어를 입력하면, “/bin/” 디렉토리 밑의 ls 프로그램을 실행시켜줌.
      - bash shell에서 vi를 입력하면, “/usr/bin/” 디렉토리 밑의 vi 프로그램을 찾아서 실행함.
      - “/bin/”, “/usr/bin/”과 같은 실행 프로그램의 위치는 “환경 변수”에 포함되어있음.
   2) 프로그래밍 기능
      - 자체적인 프로그래밍 기능을 통해 프로그램 작성 가능
      - 쉘 프로그램을 쉘 스크립트라고 부름
   3) 사용자 환경설정 기능
      - 초기화 파일을 이용해 사용자 환경을 설정

# Shell 이란?

1. 사용자와 시스템간의 인터페이스 역할을 하는 프로그램
   1) vi, gdb, gcc 와 같은 프로그램




   2) 실습을 위해 Putty로 접속을 하면, sshd 라는 원격접속 프로그램이 실행되고, bash라는 linux의 기본 shell이 실행된다.

# Shell Lab

1. 프로세스의 관리와 시그널의 제어에 대해 이해하고, 작업의 제어를 지원하는 Unix Shell program을 작성하는 것을 목표로 한다.

2. Shell Lab은 trace00 부터 trace21 까지의 Trace를 모두 수행할 수 있도록 구현되어야 한다.
   1) 테스트 프로그램 'sdriver' 를 통해 테스트할 수 있다.

3. 총 2주에 걸쳐 진행되며, 각 주 별 수행 사항은 아래와 같다.
   1) Shell Lab 1주차 : trace00 ~ 07
   2) Shell Lab 2주차 : trace08 ~ 21

# Shell Lab - 준비

1. Shell Lab 파일 복사 및 압축해제
   1) cp /home/ubuntu/shlab/shlab-handout.tar.gz ~
   2) tar xvzf ~/shlab-handout.tar.gz

2. Shell Lab 파일 구성

| 파 일 | 설 명 |
|---|---|
| **Makefile** | 쉘 프로그램의 컴파일 및 테스트 |
| **README** | 도움말 |
| **tsh.c** | Shell 프로그램의 소스코드 |
| **tshref** | Shell binary의 레퍼런스 |
| **sdriver** | Shell에 각 trace 들을 실행하는 프로그램 |
| **trace{00-21}.txt** | Shell 드라이버를 제어하는 22개의 trace |
| **myspin.c, mysplit.c, mystop.c myint.c** | trace 파일들에서 불려지는 C로 작성된 프로그램 (README 참조) |

# Shell Lab - 준비

1. tsh.c 파일 내부 구성

| 함 수 | 설 명 |
|---|---|
| **eval** | 명령을 파싱 하거나 해석하는 메인 루틴 |
| **builtin_cmd** | quit, fg, bg와 jobs 같은 built-in 명령어를 해석 |
| **waitfg** | Foreground 작업이 완료될 때 까지 대기 |
| **sigchld_handler** | SIGCHILD 시그널 핸들러 |
| **sigint_handler** | SIGINT(ctrl-c) 시그널 핸들러 |
| **sigtstp_handler** | SIGTSTP(ctrl-z) 시그널 핸들러 |

# Shell Lab – Trace (1주차)

1. sdriver를 이용하여 Trace{00-07}를 테스트 할 수 있다.
   1) 각 trace의 자세한 내용은 해당 파일을 열어서 확인할 수 있다.

# Shell Lab – 시작 하기

1. shell lab 소스파일 수정
   1) tsh.c 파일을 열어 맨 위쪽에 자신의 학번과 이름을 기입한다.
   2) tsh.c 파일 내부를 수정 또는 추가 작성하여 Shell Lab의 요구사항(각 trace 별 요구사항)을 해결해 나간다.

2. shell lab 빌드
   1) make 명령을 통해 tsh.c을 컴파일 한다.
   2) 결과로 tsh 란 실행 파일이 생성된다.
      - tsh 은 본인이 작성하여 완성된 쉘의 본체

3. shell lab 실행 및 테스트
   1) 쉘은 다음과 같이 실행한다.
      - ./tsh
      - 참고로 ./tshref 의 실행을 통해 정상적으로 완성된 쉘을 경험할 수 있다.
   2) 'sdriver'를 이용하여 생성된 쉘이 제 기능을 하는지 검사할 수 있음
      - 소스 수정 후에 항상 make한 뒤 검사

# Shell Lab - 컴파일

- shlab 빌드

1) 작성한 tsh.c를 컴파일 하여 tsh 쉘을 빌드 한다.
2) 미리 작성된 Makefile이 존재하기 때문에 make 명령으로 간단히 컴파일 할 수 있다.

```
[bOOOOOOOOO@eslab shlab-handout]$ make
gcc -Wall -O2    tsh.c   -o tsh
gcc -Wall -O2    myspin.c   -o myspin
gcc -Wall -O2    mysplit.c   -o mysplit
gcc -Wall -O2    mystop.c   -o mystop
gcc -Wall -O2    myint.c   -o myint
```

3) make clean 명령

```
[bOOOOOOOOO@eslab shlab-handout]$ make clean
rm -f ./tsh ./myspin ./mysplit ./mystop ./myint *.o *~
```

# Shell Lab – Trace 검사

## 1. sdriver 사용 방법

1) ./sdriver -t <trace number> -s ./<shell name>

- ex) tsh 쉘에서 trace00 검사
- ./sdriver –t 00 –s ./tsh

2) ./sdriver.pl -h를 통해 사용 방법과 사용 가능한 옵션을 확인 할 수 있다.

| 옵 션 | 설 명 |
|---|---|
| **-h** | 도움말 출력 |
| **-V** | 자세한 동작 과정에 대한 출력 |
| **-t <trace number>** | Trace 번호 |
| **-s <shell>** | 테스트 할 쉘 프로그램 |

# Shell Lab – Trace 검사

## 2. 레퍼런스 코드 확인

1) 완성되어있는 레퍼런스 쉘을 통해 정상 동작하는 쉘의 모습을 테스트하고 살펴볼 수 있다.

- ./sderver –t XX –s ./tshref

# Shell Lab – 쉘 실행

1. tsh 쉘의 실행
   1) ./tsh를 통해 shlab을 실행할 수 있다.
   2) 참고용으로 만들어진 레퍼런스 쉘 또한 같은 방식으로 실행 시킬 수 있다.

```
[bOOOOOOOOO@eslab shlab-handout]$ ./tsh
tsh>
```

   3) trace01의 구현 이전에는 'ctrl + d'를 통해 빠져 나와야 한다.

# Shell Lab - 작성

1. tsh.c 내부의 각 함수들을 작성한다.
   - tsh.c 파일 상단에 자신의 학번과 이름을 필수로 기입한다.

2. 쉘의 구성에 있어 핵심이 되는 함수는 eval( )이다.
   - 메인은 eval( )함수 / built-in 명령은 builtin_cmd( )함수 / … /
     ✓ 나머지 도우미 함수들을 사용하여 구성하면 된다.

```
/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
*/
void eval(char *cmdline)
{
    return;
}
```

# Shell Lab – trace00

1. trace00 : EOF(End Of File)가 입력되면 종료.

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 00 -s ./tsh
Running trace00.txt...
Success: The test and reference outputs for trace00.txt matched!
Test output:
#
# trace00.txt - Properly terminate on EOF.
#

Reference output:
#
# trace00.txt - Properly terminate on EOF.
#
```

1) trace00은 EOF가 입력되면 쉘이 종료되도록 tsh.c를 내용을 구성하면 된다.
   - EOF는 'ctrl + d'를 입력했을 때를 의미한다.
2) 하지만 해당 기능은 main( )에 구현되어 있다.
   - 따라서 구현하지 않은 tsh도 tshref의 동작과 동일하다.

```
if (feof(stdin)) { /* End of file (ctrl-d) */
    fflush(stdout);
    exit(0);
}
```

# Shell Lab – trace00

2. trace00 수행 결과 확인

1) 아래와 같이 sdriver를 이용하여 tsh의 trace00을 수행해본다.

2) 이때 tshref의 trace00을 수행한 동작과 동일하면 성공이다.

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 00 -s ./tshref
Running trace00.txt...
Success: The test and reference outputs for trace00.txt matched!
Test output:
#
# trace00.txt - Properly terminate on EOF.
#

Reference output:
#
# trace00.txt - Properly terminate on EOF.
#
```

3) 다른 방법으로는 tsh 쉘을 실행시킨 뒤 'ctrl + d'를 입력하였을 때, 종료되는지 확인하는 방법이 있다.

- 먼저 tshref에서의 동작을 먼저 살펴보는 것이 중요하다.

# Shell Lab – trace01

1. trace01 : Built-in 명령어 ‘quit’ 구현

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 01 -s ./tshref
Running trace01.txt...
Success: The test and reference outputs for trace01.txt matched!
Test output:
#
# trace01.txt - Process builtin quit command.
#

Reference output:
#
# trace01.txt - Process b
#
```

```
1 #
2 # trace01.txt - Process builtin quit command.
3 #
4 quit
```

trace01.txt

1) 쉘의 명령어 입력 창에서 'quit'을 입력하면, 쉘이 종료되도록 구현하면 된다.
2) built-in 명령어를 구현하는 방법은 다음과 같다.
   - 1. eval( )함수에서 입력 받은 명령어를 파싱한다.
   - 2. 파싱된 명령어를 builtin_cmd( )함수로 전달한다.
   - 3. 해당 명령어가 "quit"인 경우 쉘을 종료할 수 있도록 builtin_cmd( )함수를 구성한다.
3) 이 구현과정을 따라 해보면서 built-in 명령 구현하는 방법을 익혀본다.

# Shell Lab – trace01

2. eval( ) 함수에서 입력 받은 명령어를 파싱하고 builtin_cmd( )함수로 전달한다.

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];        // command 저장

    // 명령어를 parseline을 통해 분리
    parseline(cmdline, argv);
    // parsing된 명령어를 전달
    builtin_cmd(argv);

    return;
}
```

**ex) tsh > A B CD**
**argv[0][0] = A**
**argv[1][0] = B**
**argv[2][0] = C**
**argv[2][1] = D**

3. 해당 명령어가 'quit'인 경우 쉘을 종료할 수 있도록 builtin_cmd( )함수를 구성한다.

```
int builtin_cmd(char **argv)
{
    char *cmd = argv[0];

    if (!strcmp(cmd, "quit")){ /* quit command */
        exit(0);
    }

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

# Shell Lab – trace01

4. tsh.c 파일을 저장하고, make를 통해 빌드 한다.

```
[b000000000@eslab shlab-handout]$ vi tsh.c
[b000000000@eslab shlab-handout]$ make
gcc -Wall -O2    tsh.c   -o tsh
```

5. 수정된 tsh 쉘을 sdriver를 통해 제대로 구현이 되었는지 테스트해본다.
   - ✓ tshref를 테스트한 동작결과와 동일하면 성공

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 01 -s ./tshref
Running trace01.txt...
Success: The test and reference outputs for trace01.txt matched!
Test output:
#
# trace01.txt - Process builtin quit command.
#
```

6. 또한 직접 tsh 쉘을 실행시키고 ‘quit’ 명령을 입력해본다.
   - ✓ 정상적으로 종료가 되는지 확인

```
[b000000000@eslab shlab-handout]$ ./tsh
tsh> quit
[b000000000@eslab shlab-handout]$
```

# 참조 - execve

1. execve(const char *path, const char *argv[], const char *envp[])
   1) #include <unistd.h> 해야 사용 가능.
   2) 현재 프로세스를 execve 함수로 호출한 프로그램으로 교체.
      - 호출한 프로그램의 텍스트, 데이터, bss, 스택이 호출된 프로그램의 것으로 교체됨.
      - PID와 열린 파일디스크립터 등은 호출한 프로그램것을 상속받음.
   3) *path: 실행할 프로그램의 전체 경로를 입력
      - “/bin/ls”
   4) *argv[]: path에 입력한 프로그램에 넘겨줄 인자를 배열 형태로 입력.
      - char *arg[] = {“-a”, “0”};
   5) *envp[]: 환경변수를 입력.

```c
#include <unistd.h>
#include <stdio.h>

int main(void) {
    char *arg[] = {"ls", "-l", 0};
    char *env[] = {0};
    execve("/bin/ls", arg, env);
    return 0;
}
```

```
sys03@localhost:~/workspace$ ./test.out
total 32
-rw-r--r-- 1 sys03 sudo  171 Nov  4 15:40 fbprocess.c
-rwxr-xr-x 1 sys03 sudo 8576 Nov  4 15:40 fbprocess.out
-rw-r--r-- 1 sys03 sudo  153 Nov  4 17:39 test.c
-rwxr-xr-x 1 sys03 sudo 8520 Nov  4 17:39 test.out
sys03@localhost:~/workspace$
```

# Shell Lab – trace02

1. trace02 : Foreground 작업 형태로 프로그램 실행

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 02 -s ./tshref
Running trace02.txt...
Success: The test and reference outputs for trace02.txt matched!
Test output:
#
# trace02.txt 1 #
#             2 # trace02.txt - Run a foreground job that prints an environment variable
# IMPORTANT:  3 #
# traces.  Ir 4 # IMPORTANT: You must pass this trace before attempting any later
# relies on y 5 # traces.  In order to synchronize with your child jobs, the driver
OSTYPE=/usr/l 6 # relies on your shell properly setting the environment.
usr/local/gam 7
              8 ./myenv
              9 NEXT
```

trace02.txt

1) 프로그램을 foreground 형태로 실행시키면 된다.
   - 이때 실행되는 프로세스는 매개변수를 가질 수도 있고, 그렇지 않을 수도 있다.
   - 해당 테스트를 살펴보면 echo를 foreground 형태로 실행하는 것을 볼 수 있다.

2) 쉘에서 새로운 프로그램을 실행시키기 위한 방법은 다음과 같다.
   - fork( )를 통해 자식 프로세스를 생성
   - 자식 프로세스에서 execve( )를 이용해 새로운 프로그램 실행

# Shell Lab – trace02

2. 다음 코드를 참조하여 trace02을 해결하는 쉘 코드를 구현해본다.

```
void eval(char *cmdline)
{
    char *argv[MAXARGS];          // command 저장
    pid_t pid;                    // process ID

    parseline(cmdline, argv);

    if (!builtin_cmd(argv)){
        if( /* Child Process 체크 */ ){ // Child Process 인 경우, execve()수행
            if((execve(argv[0], argv, environ) < 0)){
                printf("%s : Command not found\n", argv);
                exit(0);
            }
        }
    }
    return;
}
```

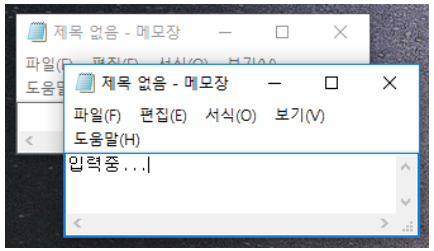
1) Hint

- fork를 통해 자식 프로세스를 생성하고, 실행되고 있는 프로세스가 자식 프로세스인지 확인.

2) execve( )사용법을 교과서에서 참고하고, 설명을 보고서에 첨부

# 참조 - Foreground 와 Background

1. 윈도우 환경에서 메모장을 2개 연달아 실행하면, 먼저 실행한 메모장은 제목이 회색으로 바뀌며 입력할 수 없다.
   1) 검은색 제목의 입력 가능한 메모장이 Foreground process
   2) 회색 제목의 입력 불가능한 메모장이 Background process




2. 리눅스의 Process도 이와 같이 Foreground proces와 Background process로 나뉜다.
   1) 5초간 실행되는 프로그램을

**foreground 실행**

```
sys03@localhost: ~/workspace
sys03@localhost:~/workspace$ ./fbprocess.out
sys03@localhost: ~/workspace
sys03@localhost:~/workspace$ ./fbprocess.out
result=5
sys03@localhost:~/workspace$ ^C
sys03@localhost:~/workspace$
```

**background 실행**

```
sys03@localhost: ~/workspace
sys03@localhost:~/workspace$ ./fbprocess.out &
[1] 22016   PID
sys03@localhost:~/workspace$ result=5

[1]+  Done                    ./fbprocess.out
sys03@localhost:~/workspace$
```

# Shell Lab – trace05

1. trace05 : Background 작업 형태로 프로그램 실행

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 05 -s ./tshref
Running trace05.txt...
Success: The test and reference outputs for trac
Test output:
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
(1) (18045) ./myspin1 &
tsh> quit
```

```
1  #
2  # trace05.txt - Run a background job.
3  #
4  /bin/echo -e tsh\076 ./myspin1 \046
5  NEXT
6  ./myspin1 &
7  NEXT
8
9  WAIT
10 SIGNAL
11
12 /bin/echo -e tsh\076 quit
13 NEXT
14 quit
```

trace05.txt

1) 프로그램을 background 형태로 실행시키면 된다.
   - 해당 테스트를 살펴보면 echo를 foreground 형태로 실행한다.
   - 이때 실행되는 프로세스는 매개변수를 가질 수도 있고, 그렇지 않을 수도 있다.

# Shell Lab – trace05

2. Background 형태로 프로그램을 실행시키려면,

1) 쉘에서 작업들을 관리하는 부분이 구현되어 있어야 한다.

- Foreground 형태인 경우, 프로세스 하나만 실행되기 때문에 작업을 따로 관리할 필요가 없다.
- 하지만 background의 경우, 여러 프로세스가 실행될 수 있으므로 작업에 대한 관리가 필수적이다.

2) 따라서 아래에 제공된 자료구조를 이용하여 구현하면 된다.

```
struct job_t {              /* The job struct */
    pid_t pid;              /* job PID */
    int jid;                /* job ID [1, 2, ...] */
    int state;              /* UNDEF, BG, FG, or ST */
    char cmdline[MAXLINE];  /* command line */
};
struct job_t jobs[MAXJOBS]; /* The job list */
/* End global variables */
```

3) 해당 자료구조는

- 프로세스의 ID(PID), 작업의 ID(JID), 프로세스의 상태(State), 사용자가 입력한 명령 정보를 가지고 있다.

4) 실행되고 있는 작업들은 이 자료구조 형태로 저장되며, jobs[MAXJOBS] 배열을 통해 관리된다.

# Shell Lab – trace05

3. 작업 관리와 관련된 함수들은 다음과 같다.

| 함 수 | 설 명 |
|---|---|
| **initjobs(struct job_t *jobs)** | 작업 리스트 초기화 |
| **addjob (struct job_t *jobs, pid_t pid, int state, char *cmdline)** | 작업 리스트에 작업을 추가 |
| **deletejob (struct job_t *jobs, pid_t pid)** | 작업 리스트에서 작업을 제거 |
| **pid2jid(pid_t pid)** | 프로세스 ID를 작업 ID로 맵핑 |
| **listjobs(struct job_t *jobs)** | 작업 리스트를 출력 |

✓ 해당 함수에 대한 자세한 동작은 코드분석을 통해 알아낼 것

# Shell Lab – trace05

4. 다음 코드 형태를 참조하여 eval( )함수를 구성하고, 쉘을 구현해본다.

```
//addjob() 도우미 함수를 이용해서 joblist에 job을 추가한다
if(/*foreground job 체크 */)
{
        //......//
}
else(/*background job 체크*/)
{
        //trace05가 요구하는 출력양식에 맞추여 출력
        //pid2jid() 도우미함수를 이용해서 양식에 맞추어 출력
}
```

1) foreground 작업인 경우
   - 자식 프로세스가 종료될 때까지 기다린다. 자식 프로세스가 종료되면 작업 리스트에서 작업을 제거한다.
     - waitpid( )함수를 활용하여, 자식 프로세스가 종료될 때까지 기다린다.

2) background 작업인 경우
   - 해당 작업의 정보를 출력하는 양식을 확인하고 해당 양식에 맞추어 출력할 수 있도록 프린트 문을 구성한다.

# Shell Lab – trace07

1. trace07 : Built-in 명령어 'jobs' 구현

```
sys03@localhost:~/workspace/shlab-handout$ ./sdriver -V -t 07 -s ./tshref
Running trace07.txt...
Success: The test and reference outp
Test output:
#
# trace07.txt - Use the jobs builtin
#
tsh> ./myspin1 10 &
(1) (20717) ./myspin1 10 &
tsh> ./myspin2 10 &
(2) (20719) ./myspin2 10 &
tsh> jobs
(1) (20717) Running ./myspin1 10 &
(2) (20719) Running ./myspin2 10 &
```

```
#
# trace07.txt - Use the jobs builtin command.
#
/bin/echo -e tsh\076 ./myspin1 10 \046
NEXT
./myspin1 10 &
NEXT

/bin/echo -e tsh\076 ./myspin2 10 \046
NEXT
./myspin2 10 &
NEXT

WAIT
WAIT

/bin/echo -e tsh\076 jobs
NEXT
jobs
NEXT

quit
```

# Shell Lab – trace07

2. trace07

1) 두 개의 background 작업을 실행한 후, built-in 명령어 'jobs'을 실행한다. 해당 명령어를 입력 받으면, 현재 실행되고 있는 작업의 리스트를 출력해준다.

2) 앞서 구현한 built-in 명령어 'quit'과 비슷한 방식으로 구현하면 된다.
   - 이때 작업 리스트를 출력하는데 사용하는 함수는 listjobs( )이다.
   - 해당 함수의 사용 방법은 소스코드 분석을 통해 알아낸다.

# Shell Lab – 예외 처리

1. 시스템 콜 호출 시, 예외가 발생하면 unix_error 함수(tsh.c에 구현되어있음)를 이용하여 오류 번호에 따른 오류 메시지를 출력할 수 있도록 한다.
   1) ex) fork( )로 프로세스 생성에 실패한 경우의 예외처리

```
/* Create a child process */
if ((pid = fork()) < 0)
    unix_error("fork error");
```

2. 쉘 프로그램에서 예외가 발생할 때, app_error 함수(tsh.c에 구현되어 있음)를 이용하여 단순한 에러 메시지를 출력할 수 있도록 한다.
   1) ex) tsh.c의 main 함수에 구현되어 있는 쉘 커멘드라인에 대한 예외처리

```
if ((fgets(cmdline, MAXLINE, stdin) == NULL) && ferror(stdin))
        app_error("fgets error");
```

# 주의 사항

1. Shell Lab을 수행하다 보면, 부모 프로세스가 자식 프로세스보다 먼저 종료되어 자식 프로세스가 좀비가 되는 경우가 있다.

   1) 쉘 종료 후, 'ps' 명령어를 입력

```
[b000000000@eslab shlab-handout]$ ps
  PID TTY          TIME CMD
 7714 pts/1    00:00:00 bash
 7805 pts/1    00:00:00 ps
```

   2) 위와 같이 tsh가 좀비가 되어 남아있는 것을 볼 수 있다. 따라서 해당 좀비 프로세스를 제거해야 한다.

   - kill -9 <PID>
   - ex) kill -9 29895

   3) 좀비 프로세스를 많이 만들면 감점! (쉘 테스트 후 실시간 체크 바람)

# 과제

1. Shell Lab
   1) trace00 ~ trace07에 대한 코드 작성
2. Shell Lab 보고서
   1) 각 trace 별, tshref를 수행한 결과와 본인이 구현한 tsh와의 동작 일치를 증명
      - sdriver 수행 결과와 tsh에서의 정상작동 모습(-v 옵션 사용)
   2) 각 trace 별, 플로우 차트
      - 간단한 수행과정을 플로우차트로 나타내면 됨.
   3) 각 trace 별, 해결 방법에 대한 설명
3. 제출
   1) shlab-handout 디렉토리를 통째로 압축
      - 파일명: [sys03]shell_학번.tar.gz
   2) 결과 보고서를 작성
      - 파일명: [sys03]shell_학번.pdf
   3) I.과 II. 두개를 하나로 압축
      - 파일명:[sys03]shell_학번_이름.zip

# 제출 사항

1. 사이버캠퍼스에 제출(추가적으로 보고서를 출력하여 서면 제출)
   1) 자세한 양식은 앞장 슬라이드 참고.
   2) 파일 제목: [sys03]shell_학번_이름.zip
   3) 반드시 위의 양식을 지켜야 함. (위반 시 감점)

2. 제출일자
   1) 사이버 캠퍼스: 2016년 11월 15일 화요일 08시 59분 59초까지
   2) 서면 제출 : 2016년 11월 15일 실습시간까지