설계과제 개요 : SoongSil Shell #3

Linux System Programming by Jiman Hong (jiman@ssu.ac.kr),
Sping 2017, School of CSE, Soongsil University

1. 개요

쉘(Shell)은 유닉스/리눅스 컴퓨팅 환경에서 기본적이고 중요한 부분으로, 명령 줄(Command line, cmd line)이라고도 부르며 사용자와 커널 사이의 인터페이스를 담당한다. 사용자의 명령을 해석하여 커널기능을 이용할 수 있게 해주고 자체적으로 프로그래밍 기능을 제공하여 반복적으로 수행해야 하는 명령어를 처리할 수 있다. 또, 각 사용자들의 환경을 저장하여 사용자에 맞는 환경을 제공한다.

Reader-Writer문제는 전산학에서 다루는 문제들 중 하나로써, 여러 명의 Reader와 Writer들이 하나의 저장 공간을 공유하며 이를 접근하는 상황에서 생겨나는 문제이다. Reader는 공유하는 저장 공간에서 데이터를 읽어 올 수 있으며, 여러 명의 Reader들이 동시에 데이터를 읽어오는 것이 가능하다. 하지만 Writer가 저장 공간에 접근하여 데이터를 쓰고 있는 경우에는 Reader와 Writer 모두 접근이 제한되어야 한다.

운영체제에서는 프로세스 및 사용자 사이의 소통을 위한 몇 가지 도구를 제공하고 있다. 몇 가지 도구들 중의 하나인 시그널은 비동기적인 사건의 발생을 통지하기 위한 용도와, 사건을 동기화시키기 위한 용도로 사용될 수 있다. 예를 들어, SIGALRM은 사건을 동기화시키기 위해서, SIGKILL, SIGSEGV는 비동기적인 사건을 통지하기 위해서 사용한다.

FIFO는 PIPE와 달리 이름이 있는 파일을 통하여 통신이 이루어지므로, 서로 다른 세션에 있는 프로세스라도 관계없이 통신을 할 수 있으며, 다중의 클라이언트를 받아들이기 위한 서버모델을 만들 수 있다. 다중의 클라이언트로부터 메시지를 받아서 처리하는 것으로 끝나는 경우에도 간단하게 구현할 수 있다는 것이 장점이다.

디몬 프로세스는 멀티태스킹 운영체제에서 사용자가 직접제어하지 않고, 백그라운드 상에서 여러 작업을 수행하는 프로그램을 뜻한다. 시스템 로그를 남기는 syslogd처럼 보통 디몬 프로세스는 끝이 'd'로 끝난다. 디몬은 대개 프로세스 트리에서 init 바로 아래에 위치한다. 디몬이 되는 방법은 일반적으로 자식 프로세스를 생성하고 자신의 수행을 마치면서 init이 고아가 된 자식 프로세스를 자기 밑으로 데려가도록 하는 방식이다.

2. 목표

새로운 명령어를 시스템 함수를 사용하여 구현함으로써 쉘의 원리를 이해하고, 유닉스/리눅스 시스템에서 제공하는 여러 시스템 자료구조를 이용하여 프로그램을 작성함으로써 시스템 프로그래밍 설계 및 응용 능력을 향상시킨다.

또한, Reader-Writer의 동기화 문제를 이해하고 이를 시그널과 FIFO파일을 이용해서 공유 파일의 접근을 제어하는 능력을 향상시킨다. 그리고 디몬 프로세스를 이해하고 디몬의 규칙에 알맞도록 구현하는 능력을 배양한다.

- 아래의 쉘 명령어 구현

명령어	내용
ssu_vim	Reader와, Writer의 역할을 수행할 프로그램이며, 상황에 알맞게 자식 프로세스로 Vim 프로세스를 생성하여 에디터를 수행하는 명령어
ssu_ofm	디몬 프로세스를 생성하며, 이 디몬 프로세스는 공유 파일의 Reader-Writer문제를 FIFO파일과 시그널을 이용해서 제어하는 관리자의 역할을 수행

3. 팀 구성

개인별 프로젝트

4. 개발환경

가. OS : **Ubuntu 16.04 32bit**

나. Tools : vi(m), gcc, gdb

5. 보고서 제출 방법

1) 제출할 파일

- **보고서와 소스파일을 함께 압축**하여 제출
  - 보고서 파일 : 한글로 작성
  - 소스코드 : Makefile 및 ssu_vim.c, ssu_ofm.c 등의 소스코드
  - 압축파일명 : [P3]_학번_v0.0.zip

2) 제출할 곳

- http://oslab.ssu.ac.kr 접속 [Courses] ⇒ [리눅스 시스템 프로그래밍] ⇒ [과제제출]
- 게시물 제목은 파일이름과 동일함
- 압축된 파일을 첨부하여 과제 제출

3) 제출 기한

- 6월 5일(월) 오후 11시 59분 59초까지 과제 게시판으로 제출

6. 보고서 양식

보고서는 다음과 같은 양식으로 작성

1. 과제 개요
2. 설계
3. 구현
   - 각 함수별 기능
4. 테스트 및 결과
   - 테스트 프로그램의 실행 결과를 분석
5. 소스코드(주석 포함)

7. 설계 및 구현

1) 전체 프로그램 수행 과정

가) **ssu_vim은 ssu_ofm 명령어로 생성되는 디몬 프로세스가 실행되기 이전에 수행될 수 없음**

나) ssu_ofm은 ssu_vim이 파일 사용을 요청할 때 전송하는 시그널을 받아서 그것들을 큐의 형태로 관리하는 관리자의 역할을 함

다) ssu_vim에서 파일 사용을 요청할 때, 요청하려는 파일이 ssu_ofm에서 관리하는 중인지 아닌지를 확인해야 함

라) ssu_vim과 ssu_ofm에서 상황을 알리는데 사용하는 시그널은 **사용자 정의 시그널인 SIGUSR1, SIGUSR2**를 사용

마) ssu_ofm은 ssu_log.txt파일을 open하여, 파일에 접근하려는 프로세스의 id, 접근이 끝난 프로세스의 id를 기본적으로 저장하며, 옵션에 따라 추가적인 정보를 저장함

바) ssu_ofm에게 SIGKILL 시그널이 전송되면(터미널에서 kill 명령어를 통해 전송) ssu_log.txt를 마무리하고 수행을 종료함

사) FIFO 파일에는 ssu_vim에서 어떤 파일을 요청했는지의 정보를 저장함

아) FIFO 파일의 이름은 ssu_vim, ssu_ofm 두 프로그램 모두 알고 있다고 가정

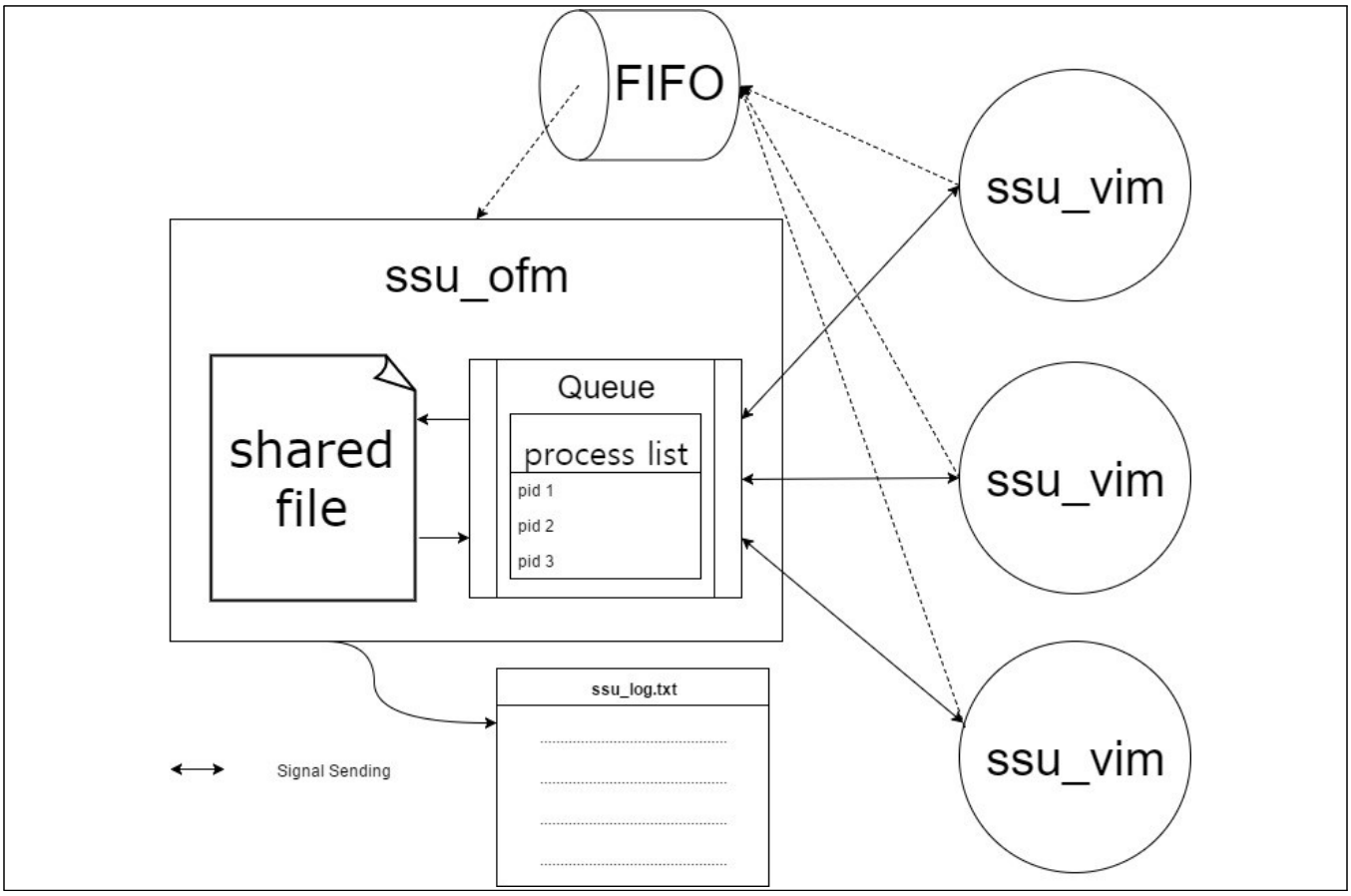

그림 1. ssu_ofm과 ssu_vim의 대략적인 실행 관계

2) ssu_vim

가) 프로그램 설명

- ssu_vim은 vim을 감싸고 있는 명령어로써, 옵션에 따라서 vim을 수행하도록 구현됨
- 기존의 vim과 다르게 읽기, 쓰기, 읽기-쓰기 옵션을 제공하며, **이들은 실행에 있어서 인자로 반드시 주어져야 함**
- 읽기 옵션은 단순히 파일의 내용을 읽어서 출력해주는 기능을 가짐
- 쓰기 옵션은 ssu_ofm의 디몬 프로세스와 시그널 전송을 통해서 vim 에디터로 수정하려는 파일에 접근하도록 구현
- 읽기-쓰기 옵션은 파일의 내용을 읽어서 출력해준 다음, 사용자에게 수정 여부를 물은 뒤, 수정할 경우에는 쓰기 옵션과 동일한 작업을 수행함

나) 프로그램 명세

- ssu_vim <FILENAME> <-r ¦ -w ¦ -rw> [OPTION]
  - <FILENAME>
    - ✓ read혹은 write할 파일의 이름
  - <-r ¦ -w ¦ -rw>
    - ✓ -r 옵션 : 읽기 옵션
      - ▶ vim 에디터를 실행시키지 않고, 파일의 내용을 표준 출력으로 보여주기만 하면 됨
    - ✓ -w 옵션 : 쓰기 옵션
      - ▶ ssu_ofm이 만든 디몬 프로세스와 시그널을 주고받으며 동기화한 다음, vim에디터를 실행함.
      - ▶ ssu_vim이 수정하려는 파일에 접근할 수 있는지 여부를 확인하기 위해서 **SIGUSR1 시그널을 사**

용함

▶ ssu_ofm으로부터 시그널을 받기 전까지는 대기 상태(무한 루프)에 접어듬.

▶ 루프 중에 1초 간격으로 Waiting for Token...<FILENAME> 문자열을 출력함.

▶ ssu_vim이 ssu_ofm으로부터 수정하려는 파일에 접근이 가능하다는 시그널인 **SIGUSR1을 받으면**, 자식 프로세스를 생성하여 vim에디터를 실행시킴

▶ ssu_vim은 자식 프로세스인 vim에디터가 수행을 마칠 때까지 대기해야 함

▶ 자식 프로세스인 vim에디터가 수행을 마치면 ssu_vim은 수행을 마쳤다고 ssu_ofm에게 **SIGUSR2를 전송**

▶ SIGUSR2 시그널을 전송하고 나서 ssu_vim은 종료

✓ -rw 옵션 : 읽기-쓰기 옵션

▶ 시작은 -r옵션과 유사하게 실행하되, 사용자에게 수정할 것인지 여부를 물어보며, 만약 수정할 경우에는 -w옵션과 동일하게 수행함

ssu_vim 실행 예시 [2-1] -rw 옵션

```
oslab@localhost:~/assign3$ ./ssu_vim test.txt -rw
Hello
Linux System Programming Assignment#3
Final Exam
Would you like to modify 'test.txt'? (yes/no) :
```

✓ 위 3개의 옵션은 동시에 사용될 수 없기 때문에 **같이 주어지는 경우에 대해서 에러 처리 구현**

- [OPTION]

✓ [OPTION]에 해당하는 옵션들은 각각 해당하는 옵션의 문자열을 출력해야 함. 또한, 읽기, 쓰기, 읽기-쓰기 옵션에 종속적이지 않음

✓ -t

▶ 최종 수정시간(Modification time)을 출력함

▶ 현재 시각을 출력함

▶ 출력을 보이기 위해서 1초 정도 실행을 잠시 멈추고 다시 수행

▶ 만약, 다른 Writer가 사용 중이어서 대기하는 경우, 문자열 출력과 같이 현재 시각을 1초 간격으로 출력

▶ 읽기, 쓰기, 읽기-쓰기 옵션에 대해서 수행을 마치고 수정이 있었는지를 확인

▶ 수정이 있었다면 'There was modification.' 문자열 출력,
그렇지 않다면 'There was no modification.' 문자열 출력

ssu_vim 실행 예시 [2-2] -t 옵션 (위에서부터 –r 옵션, vim 에디터를 q로 종료, wq로 종료한 결과)

```
oslab@localhost:~/assign3$ ./ssu_vim test.txt -r -t
##[Modification Time]##
Last Modification time of 'test.txt': [2017-05-08 00:57:23]
Current time: [2017-05-08 00:58:13]
Hello
Linux System Programming Assignment3
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -t
##[Modification Time]##
Last Modification time of 'test.txt': [2017-05-08 00:57:23]
Current time: [2017-05-08 00:58:15]
Waiting for Token...test.txt[2017-05-08 00:58:16]
##[Modification Time]##
- There was no modification.
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -t
##[Modification Time]##
Last Modification time of 'test.txt': [2017-05-08 00:57:23]
Current time: [2017-05-08 00:58:22]
Waiting for Token...test.txt[2017-05-08 00:58:23]
##[Modification Time]##
- There was modification.
oslab@localhost:~/assign3$
```

✓ -s

▶ 파일의 크기를 출력하는 옵션

▶ 쓰기, 읽기-쓰기 **옵션을 수행하기 전의 파일 크기**를 출력

▶ **수행이 끝나고 난 다음의 파일 크기**를 출력

ssu_vim 실행 예시 [2-3] -s 옵션 (위에서부터 –r 옵션, vim 에디터를 q로 종료, wq로 종료한 결과)

```
oslab@localhost:~/assign3$ ./ssu_vim test.txt -r -s
Hello
Linux System Programming Assignment3
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -s
Waiting for Token...test.txt
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -s
Waiting for Token...test.txt
##[File Size]##
-- Before modification : 43(bytes)
-- After modification : 43(bytes)
oslab@localhost:~/assign3$
```

✓ -d

▶ 이전의 파일과 수정된 파일의 어느 부분이 달라졌는지를 비교하는 옵션

▶ 이 옵션은 diff 명령어를 사용. man diff 참조

▶ 자식 프로세스를 생성하여 diff 기능을 수행하도록 구현

▶ 이 옵션은 **수정이 일어난 경우에만** 적용하도록 구현

ssu_vim 실행 예시 [2-4] -d 옵션 (위에서부터 –r 옵션, vim 에디터를 q로 종료, wq로 종료한 결과)

```
oslab@localhost:~/assign3$ ./ssu_vim test.txt -r -d
Hello
Linux System Programming Assignment3
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -d
Waiting for Token...test.txt
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -d
Waiting for Token...test.txt
##[Compare with Previous File]##
2c2
< Linux System Programming Assignment#3
---
> Linux System Programming Assignment3
oslab@localhost:~/assign3$
```

다) 세부 기능 및 기능별 요구 조건

○ 입력 요구조건

- ssu_vim에서 어떠한 경우(옵션 구현 포함)에라도 system()함수 사용 불가능
- 수행이 모두 끝나고 나서 vim으로 만든 파일 이외의 파일(로그 제외)이 존재하면 안 됨
- 쓰기 옵션 및 읽기-쓰기 옵션은 **ssu_ofm이 실행되고 있는 경우에만 수행되도록** 구현

ssu_vim 실행 예시 [2-5] ssu_vim이 실행 가능한 경우

```
oslab@localhost:~/assign3$ ps -ef | grep ssu_ofm
oslab     11534 11424  0  5월07 pts/32 00:00:07 vi ssu_ofm.c
oslab     14830 13517  0 01:06 pts/6    00:00:00 grep --color=auto ssu_ofm
oslab@localhost:~/assign3$ ./ssu_vim test.txt -r
Hello
Linux System Programming Assignment#3
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w
where is ssu_ofm?
ssu_vim error
oslab@localhost:~/assign3$
```

ssu_ofm 없음

- 시각은 시스템 상의 지역 시간을 기준으로 함

3) ssu_ofm

가) 프로그램 설명

- ssu_ofm은 디몬 프로세스를 생성하여 그 디몬 프로세스가 인자로 주어진 파일에 대해서 Reader-Writer 문제가 발생하지 않게끔 제어하는 명령어
- 추가적인 옵션에 따라 로그 파일의 내용이 추가됨
- 기본적으로 **로그 파일(ssu_log.txt)에 저장되는 정보는 요청한 프로세스 ID, 요청한 파일 이름, 그리고 수행이 끝난 프로세스 ID**를 저장함
- ssu_vim에서 보낸 **SIGUSR1을 받으면**, FIFO 파일과 시그널을 보낸 프로세스 ID를 확인함
- 공유 파일이 사용 중이 아닌 경우에는 요청한 프로세스에게 **SIGUSR1을 전송**해서 사용 가능하게 함
- 공유 파일이 사용 중인 경우에는 요청받은 프로세스를 큐에 넣어둠
- ssu_vim으로부터 **SIGUSR2 시그널을 받은 경우**(=ssu_vim이 수행을 마친 상황), 큐에 저장해둔 프로세스에게 차례로 **SIGUSR1 시그널을 전송**함

나) 프로그램 명세

- ssu_ofm <FILENAME> [OPTION]
  - <FILENAME>
    - ✓ 공유 파일로 지정할 파일 이름
  - [OPTION]
    - ✓ -l
      - ▶ ssu_vim으로부터 SIGUSR2 시그널을 받을 때 마다 ssu_log.txt와 내용은 같으나 별개의 로그파일을 생성하게 하는 옵션
      - ▶ 파일명은 **SIGUSR2를 받은 시각**인 '[연도-월-일 시:분:초]'의 형태로 생성
      - ▶ -t 옵션의 영향을 받음

ssu_ofm 실행 예시 [3-1] -l 옵션 (다른 터미널에서 쓰기 전용으로 8번 접근한 경우)

```
oslab@localhost:~/assign3$ ./ssu_ofm test.txt -l
Daemon Process Initialization.
oslab@localhost:~/assign3$ ls -al
합계 244
drwxrwxr-x  4 oslab oslab  4096  5월  8 02:05 .
drwxr-xr-x 29 oslab oslab  4096  5월  8 02:05 ..
-rw-r--r--  1 oslab oslab 53248  5월  8 02:05 .ssu_ofm.c.swp
-rw-r--r--  1 oslab oslab 40960  5월  8 01:11 .ssu_vim.c.swp
-rw-rw-r--  1 oslab oslab   224  5월  3 16:37 Makefile
-rw-rw-rw-  1 oslab oslab   373  5월  8 01:41 [2017-05-08 01:41:19]
-rw-rw-rw-  1 oslab oslab   373  5월  8 01:43 [2017-05-08 01:43:50]
-rw-rw-rw-  1 oslab oslab   273  5월  8 01:45 [2017-05-08 01:45:20]
-rw-rw-rw-  1 oslab oslab   273  5월  8 01:51 [2017-05-08 01:51:59]
-rw-rw-rw-  1 oslab oslab   418  5월  8 01:52 [2017-05-08 01:52:14]
-rw-rw-rw-  1 oslab oslab   401  5월  8 01:53 [2017-05-08 01:53:33]
-rw-rw-rw-  1 oslab oslab   546  5월  8 01:53 [2017-05-08 01:53:37]
-rw-rw-rw-  1 oslab oslab   273  5월  8 02:04 [2017-05-08 02:04:47]
-rw-rw-rw-  1 oslab oslab   401  5월  8 02:05 [2017-05-08 02:05:49]
-rwxrwxr-x  1 oslab oslab  7888  4월 17 01:12 a.out
drwxrwxr-x  2 oslab oslab  4096  4월 13 01:25 hi
-rwxrwxr-x  1 oslab oslab  7724  4월 17 11:31 inotify_test
-rw-rw-r--  1 oslab oslab  2112  4월 17 11:31 inotify_test.c
drwxrwxrwx  2 oslab oslab  4096  5월  8 00:19 ssu_dir
-rw-rw-rw-  1 oslab oslab   401  5월  8 02:05 ssu_log.txt
-rwxrwxr-x  1 oslab oslab 17472  5월  8 02:05 ssu_ofm
-rw-rw-r--  1 oslab oslab 11598  5월  8 02:05 ssu_ofm.c
-rwxrwxr-x  1 oslab oslab 17420  5월  8 02:05 ssu_vim
-rw-rw-r--  1 oslab oslab 10594  5월  8 01:11 ssu_vim.c
-rw-rw-r--  1 oslab oslab    44  5월  8 01:20 test.txt
oslab@localhost:~/assign3$ cat \[2017-05-08\ 02\:05\:49\]\
<<Daemon Process Initialized with pid : 15761>>
Initialized with Default Value : 16
Requested Process ID : 15762, Requested Filename : test.txt
Finished Process ID : 15762
oslab@localhost:~/assign3$
```

- ✓ -t
  - ▶ 로그 파일(-l 옵션 포함)에 시각을 출력하도록 하는 옵션
  - ▶ 출력과 같은 형태로 로그를 출력

ssu_ofm 실행 예시 [3-2] -t 옵션 (다른 터미널에서 쓰기 전용으로 접근한 경우)

```
oslab@localhost:~/assign3$ ./ssu_ofm test.txt -t
Daemon Process Initialization.
oslab@localhost:~/assign3$ !ps
ps -ef | grep ssu_ofm
oslab    11534 11424  0  5월07 pts/32 00:00:11 vi ssu_ofm.c
oslab    15875  1401 94 02:13 ?        00:00:09 ./ssu_ofm test.txt -t
oslab    15879 13517  0 02:14 pts/6    00:00:00 grep --color=auto ssu_ofm
oslab@localhost:~/assign3$ kill -9 15875
oslab@localhost:~/assign3$
oslab@localhost:~/assign3$ cat ssu_log.txt
[2017-05-08 02:13:55] <<Daemon Process Initialized with pid : 15875>>
Initialized with Default Value : 16
[2017-05-08 02:14:00] Requested Process ID : 15876, Requested Filename : test.txt
oslab@localhost:~/assign3$
```

✓ -n <NUMBER>

▶ NUMBER의 크기만큼 큐의 크기를 정하는 옵션

▶ 기본적으로 큐의 크기는 16을 가짐

▶ ssu_log.txt에 몇 개로 초기화 되었는지의 여부를 출력함

ssu_ofm 실행 예시 [3-3] -n 옵션

```
oslab@localhost:~/assign3$ ./ssu_ofm test.txt -n 32
Daemon Process Initialization.
oslab@localhost:~/assign3$ !ps
ps -ef | grep ssu_ofm
oslab    11534 11424  0  5월07 pts/32 00:00:12 vi ssu_ofm.c
oslab    15916  1401 99 02:17 ?        00:00:03 ./ssu_ofm test.txt -n 32
oslab    15918 13517  0 02:17 pts/6    00:00:00 grep --color=auto ssu_ofm
oslab@localhost:~/assign3$ kill -9 15916
oslab@localhost:~/assign3$
oslab@localhost:~/assign3$ cat ssu_log.txt
<<Daemon Process Initialized with pid : 15916>>
Initialized Queue Size : 32
oslab@localhost:~/assign3$
```

✓ -p <DIRECTORY>

▶ 생성되는 로그 파일을 현재 디렉터리 기준으로 저장할 새로운 디렉터리를 인자로 받음

▶ ssu_log.txt가 생성되는 경로 및 -i 옵션으로 생성되는 로그 파일의 경로에도 영향을 미침

ssu_ofm 실행 예시 [3-4] -p 옵션

```
oslab@localhost:~/assign3$ ./ssu_ofm test.txt -p ssu_dir
Daemon Process Initialization.
oslab@localhost:~/assign3$ !ps
ps -ef | grep ssu_ofm
oslab    11534 11424  0  5월07 pts/32 00:00:12 vi ssu_ofm.c
oslab    15947  1401 92 02:20 ?        00:00:14 ./ssu_ofm test.txt -p ssu_dir
oslab    15955 13517  0 02:20 pts/6    00:00:00 grep --color=auto ssu_ofm
oslab@localhost:~/assign3$ kill -9 15947
oslab@localhost:~/assign3$ ls
Makefile  a.out  hi  inotify_test  inotify_test.c  ssu_dir  ssu_log.txt  ssu_ofm  ssu_ofm.c
oslab@localhost:~/assign3$ cd ssu_dir
oslab@localhost:~/assign3/ssu_dir$ cat ssu_log.txt
<<Daemon Process Initialized with pid : 15947>>
Initialized with Default Value : 16
Requested Process ID : 15950, Requested Filename : test.txt
Requested Process ID : 15952, Requested Filename : test.txt
oslab@localhost:~/assign3/ssu_dir$
```

✓ -id

▶ 요청한 프로세스의 사용자 이름, uid, gid의 정보를 로그에 같이 출력

▶ 출력 형식은 수행결과와 동일하게 구현

ssu_ofm 실행 예시 [3-5] -id 옵션

```
oslab@localhost:~/assign3$ ./ssu_ofm test.txt -id
Daemon Process Initialization.
oslab@localhost:~/assign3$ !ps
ps -ef | grep ssu_ofm
oslab    11534 11424  0  5월07 pts/32 00:00:12 vi ssu_ofm.c
oslab    15973  1401 96 02:22 ?        00:00:11 ./ssu_ofm test.txt -id
oslab    15982 13517  0 02:22 pts/6    00:00:00 grep --color=auto ssu_ofm
oslab@localhost:~/assign3$ kill -9 15973
oslab@localhost:~/assign3$ cat ssu_log.txt
<<Daemon Process Initialized with pid : 15973>>
Initialized with Default Value : 16
Requested Process ID : 15975, Requested Filename : test.txt
User : oslab, UID : 1002, GID : 1002
Requested Process ID : 15977, Requested Filename : test.txt
User : oslab, UID : 1002, GID : 1002
oslab@localhost:~/assign3$
```

다) 세부 기능 및 기능별 요구 조건

- ssu_ofm은 ssu_vim보다 항상 먼저 실행되고 있는 상황이라고 가정하고 구현해도 됨
- ssu_ofm은 **반드시 디몬 프로세스를 생성하는 형태로 구현되어야 함**
- 공유 파일을 요청한 순서대로 프로세스들을 스케줄링함. 즉, First-In First-Out으로 서비스 받도록 구현
- ssu_ofm에서 어떠한 경우(옵션 구현 포함)에라도 system()함수 사용 불가능
- 시각은 시스템 상의 지역 시간을 기준으로 함

4) 실행 예

전체 프로그램 실행 예시 [4-1] - ssu_ofm에서 여러 옵션을 주고 실행. 쓰기 옵션으로 2번 접근한 상황

```
oslab@localhost:~/assign3$ ./ssu_ofm test.txt -t -l -p ssu_dir -id
Daemon Process Initialization.
oslab@localhost:~/assign3$ ls ssu_dir
[2017-05-08 02:45:26]   [2017-05-08 02:45:28]   ssu_log.txt
oslab@localhost:~/assign3$ cat ssu_dir/\[2017-05-08\ 02\:45\:26\]\
[2017-05-08 02:45:22] <<Daemon Process Initialized with pid : 16162>>
Initialized with Default Value : 16
[2017-05-08 02:45:24] Requested Process ID : 16163, Requested Filename : test.txt
User : oslab, UID : 1002, GID : 1002
[2017-05-08 02:45:26] Finished Process ID : 16163
oslab@localhost:~/assign3$
```

전체 프로그램 실행 예시 [4-2] - ssu_vim에서 여러 옵션을 주고 실행

```
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -t -s -d
##[Modification Time]##
Last Modification time of 'test.txt': [2017-05-08 02:45:26]
Current time: [2017-05-08 02:49:42]
Waiting for Token...test.txt[2017-05-08 02:49:43]
##[Modification Time]##
- There was modification.
##[File Size]##
-- Before modification : 44(bytes)
-- After modification : 55(bytes)
##[Compare with Previous File]##
3d2
< Final Exam
oslab@localhost:~/assign3$
```

전체 프로그램 실행 예시 [4-3] - ssu_vim끼리 공유파일에 접근하는 상황(이미 다른 프로세스가 사용 중)

```
oslab@localhost:~/assign3$ ./ssu_vim test.txt -w -t
##[Modification Time]##
Last Modification time of 'test.txt': [2017-05-08 02:50:27]
Current time: [2017-05-08 02:57:41]
Waiting for Token...test.txt[2017-05-08 02:57:42]
Waiting for Token...test.txt[2017-05-08 02:57:43]
Waiting for Token...test.txt[2017-05-08 02:57:44]
Waiting for Token...test.txt[2017-05-08 02:57:45]
Waiting for Token...test.txt[2017-05-08 02:57:46]
```

※ 과제 구현에 필요한 함수들

1. signal()

- 리눅스 시스템에서 시그널 처리를 설정
- 1. 기존의 방법을 따를지, 2. 시그널을 무시할지, 3. 지정된 함수를 실행하게 할 것인지를 등록
- 지정된 함수는 반드시 sighandler_t의 형태를 가진 함수이어야 함

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

인자	signum	시그널 번호를 가짐
	handler	SIG_DFL(디폴트 시그널 핸들러), SIG_IGN(무시) 혹은 지정된 함수를 가짐
반환값	sighandler_t	호출 이전의 signum에 대한 시그널 핸들러
	SIG_ERR	에러 발생시 반환하며, errno가 설정됨

signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void (*old_handler)(int);

void sigint_handler(int signo) {
    printf("sigint caught!\n");
    signal(SIGINT, old_handler);
}

int main(void)
{
    old_handler = signal(SIGINT, sigint_handler);
    while(1) {
        printf("Hi\n");
        sleep(1);
    }

    exit(0);
}
```

실행결과

```
oslab@localhost:~$ ./signal_example
Hi
Hi^C
```

```
sigint caught!
Hi
Hi
^C
oslab@localhost:~$
```

2. sigaction()

- signal()보다 향상된 기능의 시그널 처리를 결정하는 함수
- sigaction()은 struct sigaction 구조체 값을 사용하므로 다양한 지정이 가능

```
struct sigaction {
    void (*sa_handler)(int);    // 시그널을 처리하기 위한 핸들러. SIG_DFL, SIG_IGN 또는 핸들러 함수
    void (*sa_sigaction)(int, siginfo_t *, void *); // 밑의 sa_flags가 SA_SIGINFO일때
                                                    // sa_handler대신에 동작하는 핸들러
    sigset_t sa_mask;            // 시그널을 처리하는 동안 블록할 시그널 집합의 마스크
    int sa_flags;                // 아래 표 설명 참조
    void (*sa_restorer)(void);   // 사용 하지 않음
}
```

sa_flag 옵션	의미
SA_NOCLDSTOP	signum이 SIGCHILD일 경우, 자식 프로세스가 멈추었을 때, 부모 프로세스에 SIGCHILD가 전달되지 않음
SA_ONESHOT	시그널을 받으면 설정된 행도을 취하고 시스템 기본 설정인 SIG_DFL로 재설정됨
SA_RESTART	시그널 처리에 의해 방해 받은 시스템 호출은 시그널 처리가 끝나면 재시작
SA_NOMASK	시그널을 처리하는 동안에 전달되는 시그널은 블록되지 않음
SA_SIGINFO	이 옵션이 사용되면 sa_handler대신에 sa_sigaction이 동작되며, sa_handler보다 더 다양한 인수를 받을 수 있음. sa_sigaction이 받는 인수에는 시그널 번호, 시그널이 만들어진 이유, 시그널을 받는 프로세스의 정보가 있음

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

인자	signum	시그널 번호
	*act	새롭게 설정할 행동
	*oldact	함수 호출 전까지의 주어진 시그널 번호에 대한 행동 정보 반환
반환값	성공 시 0, 실패 시 –1이 반환 됨	

sigaction_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

struct sigaction act_new;
struct sigaction act_old;

void sigint_handler(int signo) {
    printf("sigint caught!\n");
    sigaction(SIGINT, &act_old, NULL);
}

int main(void)
{
    act_new.sa_handler = sigint_handler;
    sigemptyset(&act_new.sa_mask);
```

```
    sigaction(SIGINT, &act_new, &act_old);
    while(1) {
        printf("Hi\n");
        sleep(1);
    }
    exit(0);
}
```

실행결과

```
oslab@localhost:~$ ./sigaction_example
Hi
Hi^C
sigint caught!
Hi
Hi
^C
oslab@localhost:~$
```

3. time(), localtime()

- time()
  - 이 함수가 반환하는 시간은 1970년 1월 1일 00:00:00부터 지금까지의 시간을 초 단위로 환산한 것임
  - 인자로 받는 변수의 주소에 반환하는 시간 값이 저장됨
- localtime()
  - time_t 값을 이용하여 지역 시간을 기준으로 값을 나눈 것을 struct tm 구조체에 저장하고 그 주소를 반환하는 함수
  - 현재 시간을 구해야 할 경우에 주로 사용
  - 반환 되는 tm 구조체의 포인터는 정적으로 할당된 메모리를 가리키고 있으므로, 추가적인 localtime() 호출이 발생하면 값이 바뀌므로 유의해서 사용해야 함

```
#include <time.h>
time_t time(time_t *t);
```

인자	*t	NULL이 아닐 경우, 가리키는 메모리에 반환하는 시간 값이 저장됨
반환값	성공 시 현재까지 흐른 시간을 초 단위로 반환, 실패 시 -1반환	

```
#include <time.h>
struct tm* localtime(const time_t *timer);
```

인자	*timer	시간 값이 저장된 time_t형 포인터
반환값	성공 시 값이 저장된 tm 구조체의 포인터 반환, 실패 시 NULL 반환	

localtime_example.c

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t rawtime;
    struct tm *timeinfo;

    time(&rawtime);
```

```
    timeinfo = localtime(&rawtime);
    printf("Current local time and date: %s", asctime(timeinfo));

    exit(0);
}
```

실행결과

```
oslab@localhost:~$ ./localtime_example
Current local time and date: Sun May 7 17:25:00 2017
```

4. mkfifo()

- pipe()에서 생성한 파이프는 부모-자식 프로세스 관계에서만 사용가능 하나, FIFO를 사용하면 서로 다른 프로세스에서도 파이프를 사용할 수 있음
- mkfifo()로 FIFO파일을 생성하면 이 파일을 파이프로 사용할 수 있음. 이 때, 이미 동일한 이름의 FIFO파일이 존재하면 에러가 발생하므로 삭제하고 사용해야 함

```
#include <sys/stat.h>
#include <sys/types.h>
int mkfifo(const char *pathname, mode_t mode);
```

인자	pathname	파이프로 사용할 파일 이름
	mode	FIFO 파일에 대한 접근 권한
반환값	성공 시 0, 실패 시 –1을 반환하고 errno가 설정됨	

main_receiver.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define  FIFO_FILE   "/tmp/fifo"
#define  BUFF_SIZE   1024

int main(void)
{
   int   counter = 0;
   int   fd;
   char  buff[BUFF_SIZE];

   if (mkfifo(FIFO_FILE, 0666) < 0) {
      fprintf(stderr, "mkfifo() error\n");
      exit(1);
   }

   if ((fd = open(FIFO_FILE, O_RDWR)) < 0) {
      fprintf(stderr, "open() error\n");
      exit(1);
   }

   while(1) {
```

```
      memset(buff, 0, BUFF_SIZE);
      read(fd, buff, BUFF_SIZE);
      printf("%d: %s\n", counter++, buff);
   }
   close(fd);
   exit(0);
}
```

main_sender.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>

#define  FIFO_FILE   "/tmp/fifo"

int main(void)
{
   int   fd;
   char *str   = "Hi FIFO!";

   if ((fd = open(FIFO_FILE, O_WRONLY)) < 0) {
      fprintf(stderr, "open() error\n");
      exit(1);
   }

   write(fd, str, strlen(str));
   close(fd);
   exit(0);
}
```

실행결과

```
oslab@localhost:~$ gcc main_receiver.c -o receiver
oslab@localhost:~$ gcc main_sender.c   -o sender
oslab@localhost:~$ ./receiver &
oslab@localhost:~$ ./sender
oslab@localhost:~$ 0: Hi FIFO!
./sender
oslab@localhost:~$ 1: Hi FIFO!
./sender
oslab@localhost:~$ 2: Hi FIFO!
./sender
oslab@localhost:~$ 3: Hi FIFO!
```

5. getpwnam(), getpwuid()

- /etc/passwd 파일의 내용을 읽어서 구조체의 포인터를 반환하는 함수
- uid, gid, 사용자 이름등의 정보를 가지고 있음

#include <pwd.h> #include <sys/types.h> struct passwd *getpwnam(const char *name);		
인자	name	사용자의 로그인 이름
struct passwd *getpwuid(uid_t uid);		
인자	uid	사용자 ID
반환값	성공 시 포인터, 실패 시 NULL을 반환하고 errno가 설정됨	

getpwnam_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pwd.h>

int main(void)
{
    struct passwd *pw_passwd;
    char path_name[128];

    printf("Enter ID :");
    scanf("%s", temp_name);

    if ((pw_passwd = getpwnam(temp_name)) == NULL) {
        fprintf(stderr, "getpwnam error\n");
        exit(1);
    }

    printf("ID\t: %s\nPID\t: %d\nGID\t: %d\n", pw_passwd -> pw_name, pw_passwd -> pw_uid, pw_passwd -> pw_gid);
    exit(0);
}
```

실행결과

```
root@localhost:/home/oslab# ./ssu_getpwnam
Enter ID: root
ID      : root
PID     : 0
GID     : 0
```

※ 과제 수행 시 유용하게 사용할 수 있는 Tips

1. 프로세스 관련 커맨드 라인 명령어

- ps
  - 현재 시스템에서 활성화 된 프로세스의 리스트이며 PID, TTY, 실행 시간등의 정보를 확인 할 수 있음
  - -ef 옵션과 같이 사용하면 현재 시스템에서 수행중인 모든 프로세스를 출력
  - 자세한 내용은 man ps를 참조
- kill
  - 프로세스에게 시그널을 전송하는 명령어
  - 인자로 아무것도 주지 않았다면 기본적으로 SIGTERM이 전송
  - 강제로 프로세스를 종료시키려면 -9(SIGKILL) 옵션을 사용

실행결과
oslab@localhost:~/lsp$ ps   PID TTY   TIME CMD 9745 pts/11   5-15:46:50 fd_read 10100 pts/11   00:00:00 bash 14190 pts/11   00:00:00 ps oslab@localhost:~/lsp$ kill -9 9745 oslab@localhost:~/lsp$ [1]+ 죽었음   ./fd_read

8. 설계 구성 요소

1) 목표 설정
   - 주어진 요구 조건을 이해하고 명확한 설계 목표를 설정한다.
   - 설계의 목표 및 요구조건을 문서화한다.
2) 분석
   - 목표 설정에서 명시한 요구 조건을 분석하고 해결을 위한 기본 전략을 수립한다.
   - 문제 해결을 위한 배경 지식을 이론 강의에서 듣고 개별적으로 학생들이 다양한 경로를 통해 자료를 찾아 분석한다.
3) 합성 (구조 설계)
   - 분석 결과를 토대로 적절한 구조를 도출한다.
   - 도출된 구조를 토대로 모듈을 작성한다.
4) 제작 (구현)
   - 각 모듈의 입출력을 명세한다.
   - 구조와 모듈을 C 언어로 구현하고 이를 컴파일하여 실행 파일을 만든다.
5) 시험 및 평가(성능 평가)
   - 모듈의 입출력 명세에 따라, 다양한 입력 데이타를 작성하고 모듈을 테스트한다.
   - 작성된 실행 파일이 안정적으로 수행되는지 다양하게 테스트하고 이를 평가한다.
6) 결과 도출
   - 안정적으로 수행되는 최종 결과(Output)를 캡쳐하여 최종 결과물은 보고서에 반영한다.

9. 평가 도구

1) 설계 보고서 평가
2) 실행 평가 (컴파일 및 실행)

10. 평가 준거(방법)

1) 평가 도구 1)에 대한 평가 준거
   - 소스 코드 분석 및 새로운 모듈의 설계가 제대로 이루어졌는가?
     - 설계 요구 사항을 제대로 분석하였는가?
     - 설계의 제약 조건을 제대로 반영하였는가?
     - 설계 방법이 적절한가?
   - 문서화
     - 소스 코드에 주석을 제대로 달았는가?
     - 설계 보고서가 잘 조직화되고 잘 쓰여졌는가?
2) 평가 도구 2)에 대한 평가 준거

- 요구조건에 따라 올바르게 수행되는가?

11. 기타

1) 보고서 제출 마감은 제출일 자정까지
2) 지연 제출 시 감점
   - 1일 지연 시 마다 30% 감점
   - 3일 지연 후부터는 미제출 처리
3) 압축 오류, 파일 누락
   - 50% 감점 처리 (추후 확인)
4) copy 발견시
   - F 처리

12. 점수 배점

대분류	구현 항목	배점
전체 프로그램	**점수 합**	**6**
	1. Makefile 사용	1
	2. FIFO 파일 사용	5
ssu_vim	**점수 합**	**39**
	3. -r 옵션 구현	7
	4. -w 옵션 구현	18
	5. -rw 옵션 구현	8
	6. -t 옵션 구현	2
	7. -s 옵션 구현	2
	8. -d 옵션 구현	2
ssu_ofm	**점수 합**	**55**
	9. 디몬 프로세스 구현	13
	10. 로그 파일 생성 모듈 구현	10
	11. SIGUSR1 시그널 핸들러 구현	9
	12. SIGUSR2 시그널 핸들러 구현	9
	13. -l 옵션 구현	3
	14. -t 옵션 구현	3
	15. -n 옵션 구현	3
	16. -p 옵션 구현	3
	17. -id 옵션 구현	2

▶ 필수적으로 구현해야 할 항목 : 1, 2, 3, 4, 5, 9, 10, 11, 12