

---

# **System Programming Lab**

## **Alternative Tasks**

학과 : 컴퓨터공학과

학번 : 2015313754

이름 : 김태형

<div> <div>제목</div> <div><b>AWS</b> 인스턴스 및 원격접속 결과보고서</div> </div>	<div> <div>소제목</div> <div>과제물 <b>1</b></div> </div>
--	---

## 목 차

1. SIGNALS.....	4
-----------------	---

<div> <div>제목</div> <div>AWS 인스턴스 및 원격접속 결과보고서</div> </div>	<div> <div>소제목</div> <div>과제물 1</div> </div>
---	--

## 그림 목차

[그림 1] line 11 to line 26 of myshellcommandsig.c .....	4
[그림 2] [CTRL] +[c]를 입력해도 프로그램이 종료되지 않는 화면.....	7
[그림 3] line 28 to line 68 of myshellcommandsig.c .....	8
[그림 4] shellcommand함수를 한번 이상 실행한 뒤, SIGINT에 대한 반응.....	10
[그림 5] line 70 to line 80 of myshellcommandsig.c .....	12
[그림 6] line 82 to line 100 of myshellcommandsig.c .....	14
[그림 7] 정상적인 Command를 입력했을 때 실행 화면.....	15
[그림 8] 비정상적인 Command를 입력했을 때 실행화면 .....	16

## 1. Signals

[Figure 1] is a part of myshellcommandsig.c file. Answer the following questions.

```
11  pid_t childPid;
12  int status_new; // for the satus of the child process
13  sigset_t blockMask, origMask;
14  struct sigaction salgnore, saOrigQuit, saOrigInt, saDefault;
15  int savedErrono;
16
17  sigemptyset(&blockMask);
18  sigaddset(&blockMask, SIGCHLD);
19  sigprocmask(SIG_BLOCK, &blockMask, &origMask);
20
21  salgnore.sa_handler = SIG_IGN;
22  salgnore.sa_flags = 0;
23
24  sigemptyset(&salgnore.sa_mask);
25  sigaction(SIGINT, &salgnore, &saOrigInt);
26  sigaction(SIGQUIT, &salgnore, &saOrigQuit);
```

[그림 1] line 11 to line 26 of myshellcommandsig.c

- 1) Explain what lines 17 through 19 mean in Figure 1.(10 points)

Linux에서는 signal 여러 개를 하나로 묶어서 보낼 수 있도록 Signal set 을 제공합니다.

13 line 에서

Sigset\_t blockMask, origMask ;

코드로 signal set 을 선언합니다.

17 line 에서

```
sigemptyset(&blockMask);
```

호출을 사용하여, blockMask signal set 을 아무 signal 도 없는 signal set 으로 초기화합니다

이은 18 line 에서,

```
sigaddset(&blockMask, SIGCHLD);
```

blockMask signal set 에 SIGCHLD signal 을 추가합니다.

SIGCHLD 는 Child Process 가 멈추거나 종료되었음을 Parent Process 에게 알리는 Signal 입니다. 이 Signal 을 받은 Parent Process 의 default action 은 Ignore 이므로, parent Process 를 계속 진행합니다.

18 line 에서

```
sigprocmask(SIG_BLOCK, &blockMask, & origMask);
```

sigprocmask 함수는 현재 프로세스가 Block 할 Signal 들을 저장하는 Signal mask 를 수정합니다. Signal 을 Block 한다는 의미는, OS Kernel 에게 Signal 을 Hold 하고 있다가, 나중에 Deliver 하라고 말하는 것과 같습니다. Sigprocmask 함수의 원형은 다음과 같습니다.

```
int sigprocmask (int how, const sigset_t *restrict set, sigset_t *restrict oldset)
```

첫번째 인자인 how 에는 SIG\_BLOCK, SIG\_UNBLOCK, SIG\_SETMASK 가 입력 될 수 있고, set 과 oldset 인자에는 Signal Set 이 입력될 수 있습니다. 18 line 에서는 how 인자에 SIG\_BLOCK 을 입력했는데, 이 의미는 두번째 인자인 set signal mask 에 있는 signal 들을 Signal Mask 에 추가하고, 본래의 Signal Mask 에 있는 Signal 들을 Oldset Signal Set 에 입력하는 것입니다. 즉, 18 line 으로 인해, Signal Mask 에는 SIGCHLD Signal 이 추가되고, 본래의 Signal Mask 에 있던 Signal 들은 Oldset Signal Set 에 입력되게 됩니다.

따라서, 이 process 에 SIGCHLD Signal 이 전달되어도 block 됩니다.

2) Explain what lines 21 through 26 mean in Figure 1. (10 points)

```
21  saIgnore.sa_handler = SIG_IGN;
22  saIgnore.sa_flags=0;

24  sigemptyset(&saIgnore.sa_mask);
25  sigaction(SIGINT,&saIgnore,&saOrigInt);
26  sigaction(SIGQUIT,&saIgnore,&saOrigQuit);
```

이 Line 들에서는 특정 Signal 에 대한 Action 을 sigaction 함수를 사용하여 설정합니다.

Sigaction 함수의 원형은 다음과 같습니다

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

두번째, 세번째 인자로 입력되는 `sigaction` 구조체는 다음과 같이 정의되어 있습니다.

```
struct sigaction
{
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
}
```

`Sigaction`의 첫번째 인자인 `signum`은 해당 `sigaction` 함수를 사용하여 Action을 설정할 Signal을 지정합니다. 지정한 Signal에 대한 action을 두번째 인자인 `act` `Sigaction` 구조체 변수를 사용하여 설정하고, 해당 Signal에 대한 본래의 action을 세번째 인자인 `oldact` `Sigactoin` 구조체 변수에 저장합니다.

25Line은 `SIGINT` Signal에 대한 Action을 설정하고, 26 Line은 `SIGQUIT`에 대한 Action을 설정합니다. `SIGINT` Signal은 키보드로부터 오는 인터럽트 시그널로, Signal을 받은 Process의 실행을 중지합니다. 키보드로 `[CTRL] + [c]`을 입력했을 때 보내지는 시그널입니다. 이 Signal을 받은 Process의 Default action은 Termination(종료)입니다. `SIGQUIT` Signal은 키보드로부터 오는 실행 중지 시그널로, Signal을 받은 Process의 프로세스를 종료시킨 뒤, 코어를 덤프합니다. 이 Signal을 받은 Process의 Default Action은 Core Dump입니다. 키보드로 `[CTRL] + [\]`를 입력했을 때 보내지는 시그널입니다.

21, 22, 24Line에서 `saIgnore`를 사용해서 어떤 Action을 설정했는지 서술하겠습니다.

`sigaction` 구조체의 `sa_hadler`는 `signum` 번호를 가지는 signal이 발생했을 때 실행된 함수를 설정합니다. 21line에서 `SIG_IGN`으로 설정했으므로, `signum` 번호를 가지는 해당 signal이 발생했을 때, 무시하게 됩니다.

`Sigaction`의 `sa_flags`를 통해서 시그널 처리 프로세스의 행위를 수정하는 플래그들을 명시할 수 있습니다.

`SA_NOCLDSTOP`, `SA_ONESHOT`, `SA_RESETHAND`, `SA_RESTART`, `SA_NOMASK`,  
`SA_NODEFER`, `SA_SIGINFO`

<div> <div>제목</div> <div>AWS 인스턴스 및 원격접속 결과보고서</div> </div>	<div> <div>소제목</div> <div>과제물 1</div> </div>
---	--

의 값들을 OR 연산을 통하여 원하는 플래그로 설정할 수 있습니다.

22Line에서는 0으로 설정하여 시그널 처리 프로세스의 행위를 수정하지 않았습니다.

Sigaction의 sa\_mask 멤버를 통해서 signal handler의 수행 중, Block되어야 하는 signal을 설정할 수 있습니다. 24 Line에서 sa\_mask를 empty signal set으로 만들었으므로, 해당 코드에서는 Block되어야 하는 Signal을 지정하지 않았습니다.

이렇게 설정한 Action을 25,26 Line에서 SIGINT, SIGQUIT Signal에 sigaction 함수를 사용하여 설정하였으므로, SIGINT, SIGQUIT Signal을 전달 받았을 때, 21 Line으로 인해 무시하게 됩니다. 또한, 22Line으로 인해 signal의 행위를 수정하지 않았고, 24Line으로 인해 signal handler의 수행 중 Block되어야 하는 Signal이 없게 됩니다. 따라서, 해당 프로세스를 실행할 때, [CTRL] + [c]를 입력하거나, [CTRL] + [\]를 입력해도, 프로세스가 Signal을 무시하므로, 종료되지 않고, 계속 실행됩니다. 그리고 SIGINT, SIGQUIT Signal에 본래 설정되어있던 action은 saOrigInt와 saOrigQuit에 저장됩니다.

```

Command: ^C
Parent PID is 2635
Child PID is 2639
Parent PID is 2635
exited, status=0
shellcommand()returned: status = 0

```

[그림 2] [CTRL] + [c]를 입력해도 프로그램이 종료되지 않는 화면.

[Figure 3] is a part of myshellcommandsig.c file. Answer the following questions.

```

28     switch (childPid = fork()) {
29         case -1: //error
30             return -1;
31         case 0: //for child process
32             saDefault.sa_handler = SIG_DFL;
33             saDefault.sa_flags = 0;
34             sigemptyset(&saDefault.sa_mask);
35
36             if (saOrigInt.sa_handler != SIG_IGN)
37                 sigaction(SIGINT, &saDefault, NULL);
38
39             if (saOrigQuit.sa_handler != SIG_IGN)
40                 sigaction(SIGQUIT, &saDefault, NULL);
41
42             sigprocmask(SIG_SETMASK, &origMask, NULL);
43
44             printf("Child PID is %ld\n", (long) getpid());
45             printf("Parent PID is %ld\n", (long) getppid());
46
47             execl("/bin/sh", "sh", "-c", command, (char *) NULL);
48             _exit(127);
49         default: // for parent process
50             printf("Parent PID is %ld\n", (long) getpid());
51             if (waitpid(childPid, &status_new, 0) == -1) {
52                 if (errno != EINTR) {
53                     status_new = -1;
54                     break;
55                 }
56             }
57
58             if (WIFEXITED(status_new)) { // in case of exited
59                 printf("exited, status=%d\n", WEXITSTATUS(status_new));
60             } else if (WIFSIGNALED(status_new)) { // in case of killed by signal
61                 printf("killed by signal %d\n", WTERMSIG(status_new));
62             } else if (WIFSTOPPED(status_new)) { // in case of stopped by signal
63                 printf("stopped by signal %d\n", WSTOPSIG(status_new));
64             } else if (WIFCONTINUED(status_new)) { // in case of continued
65                 printf("continued\n");
66             }
67             return status_new;
68     } // end of switch

```

[그림 3] line 28 to line 68 of myshellcommandsig.c

3) Explain what lines 32 through 34 mean in Figure 3. (10 points)

32 부터 48 까지의 Line 들은 코드를 실행하는 process 가 child process 일때만 실행됩니다.



```
32     saDefault.sa_handler=SIG_DFL;
33     saDefault.sa_flags=0;
34     sigemptyset(&saDefault.sa_mask);
```

saDefault 는 2)의 saIgnore 와 마찬가지로 sigaction 구조체 변수입니다. saDefault 변수의 멤버들에 값을 설정하여 signal 에 대한 action 을 설정합니다.

32Line 의 sa\_handler 멤버의 값을 SIG\_DFL 로 설정함으로써 signal 이 기존의 action 을 취하도록 합니다. Sa\_flag 를 0 으로 설정하여 signal 의 action 을 수정하지 않고, sa\_mask signal mask 에 Signal 이 없도록 34 Line 에서 sigemptyset 을 호출하였습니다. 즉, saDefault 의 설정을 입력 받는 signal 은 기존의 action 을 그대로 취할 것입니다.

4) Explain what lines 36 through 42 mean in Figure 3. (10 points)

```
36     if(saOrigInt.sa_handler!=SIG_IGN)
37         sigaction(SIGINT,&saDefault,NULL);
38
39     if(saOrigQuit.sa_handler!=SIG_IGN)
40         sigaction(SIGQUIT,&saDefault,NULL);
```

saOrigInt 와 saOrigQuit 은 25,26Line 에서 값이 설정된 sigaction 구조체 변수입니다.

```
25     sigaction(SIGINT,&saIgnore,&saOrigInt);
26     sigaction(SIGQUIT,&saIgnore,&saOrigQuit);
```

25,26Line 의 Sigaction 함수 call 에 의해서, saOrigInt 변수에는 SIGINT signal 에 대한 기존의 설정이, saOrigQuit 변수에는 SIGQUIT Signal 에 대한 기존의 설정이 저장됩니다. 28Line 의 fork 함수 call 로 인해 생성되고 실행된 Child Process 는 Parent Process 의 값을 그대로 가져오므로, Parent Process 에서 저장했던 saOrigInt, saOrigQuit 과 값이 동일합니다.

36,37Line 의 의미는 saOrigInt 변수의 sa\_handler 가 SIG\_IGN 이 아니었다면, SIGINT signal 에 대한 action 을 saDefault 에 설정된 action 으로 저장하는 것입니다. saOrigInt 변수는 기존의 SIGINT Signal 에 대한 Action 설정을 저장하고 있으므로, 원래 설정이 SIGINT 에 대해서 무시하는 것(SIG\_IGN)이 아니었다면, 37Line 을 실행하게 될 것입니다. saDefault 에 설정된 action 은 3)에서 서술했듯이, signal 이 기존의 action 을 그대로 취하는 것이므로, 37Line 을 실행한 Process 는 SIGINT Signal(키보드 [CTRL] +[c] 입력) 을 입력 받는다면, 기존의 설정대로 Process 를 종료할 것입니다.

39,40Line 도 마찬가지로, saOrigQuit 변수의 sa\_handler 가 SIG\_IGN 이 아니었다면, SIGQUIT signal 에 대한 action 을 saDefault 에 설정된 action 으로 저장하는 것입니다. saOrigQuit 변수는 SIGQUIT Signal 에 대한 기존의 Action 설정을 저장하고 있으므로, 원래 설정이 SIGQUIT 에 대해 무시하는 것(SIG\_IGN)이 아니었다면, 40 Line 을 실행하게 될 것입니다. 37Line 과 마찬가지로, signal action 이 default 로 설정되므로, 40Line 을 실행한 Process 는 SIGQUIT signal(키보드 [CTRL] + [\] 입력)을 받는다면, 기존의 설정대로 프로세스를 종료 시킨 뒤, 코어 덤프를 하게 됩니다.

36~40 Line 으로 인해, Parent Process 의 본래 saOrigInt 와 saOrigQuit 의 sa\_handler 가 SIG\_IGN 으로 설정되지 않았다면, 37, 40Line 으로 인해 SIGINT, SIGQUIT 에 대한 sa\_handler 가 SIG\_DFL 로 설정됩니다. 반대로, Parent Process 의 본래 saOrigInt 와 saOrigQuit 의 sa\_handler 가 SIG\_IGN 으로 설정되었다면, 37, 40Line 이 실행되지 않으므로 SIGINT, SIGQUIT 에 대한 sa\_handler 를 SIG\_IGN 으로 유지합니다. (25,26 Line 에의해) 따라서, 프로그램을 처음 실행하고 shellcommand 함수를 처음 실행하면서 Child Process 가 실행 될 때 SIGINT, SIGQUIT 을 입력하면 sa\_handler 가 SIG\_DFL 로 설정되었으므로, 프로세스가 종료됩니다. 하지만, 프로그램을 실행하고 한번 이상의 shellcommand 함수를 실행해서 parent Process 의 본래 saOrigInt 와 saOrigQuit 의 sa\_handler 가 SIG\_IGN 으로 설정되었다면, Parent Process 와 Child Process 에서 SIGINT, SIGQUIT 을 입력 받아도 프로세스를 종료하지 않습니다.

```
(base) taehyung@ubuntu:~/sys/final$ ./a.out
Command:pwd
Parent PID is 3138
Child PID is 3139
Parent PID is 3138
/home/taehyung/sys/final
exited, satus=0
shellcommand()returned: staus = 0
Command:^C
Parent PID is 3138
Child PID is 3150
Parent PID is 3138
exited, satus=0
shellcommand()returned: staus = 0
Command:
```

[그림 4] shellcommand 함수를 한번 이상 실행한 뒤, SIGINT 에 대한 반응

```
42      sigprocmask(SIG_SETMASK,&origMask,NULL);
```

origMask 는 sigset\_t 변수입니다. origMask 는 19Line 에서 값이 설정 되었습니다.

```
19      sigprocmask(SIG_BLOCK, &blockMask, & origMask);
```

19Line 에 의해 Block 된 signal 들을 origMask 에 저장했습니다. 이 저장된 값을 그대로 42Line 의 sigprocmask 함수에 전달합니다.

Sigprocmask 의 첫번째 인자인 how 가 SIG\_SETMASK 일때에는, 두번째 인자에 있는 Signal set 의 signal 들을 그대로 Block 된 signal 로 설정합니다.

즉, 42Line 으로 인해, Child Process 의 Blocked signal 들은 19Line 에서 변경하기 전의 Parent process 의 Blocked signal 과 동일하게 설정됩니다.

5) Explain what lines 58 through 66 mean in Figure 3. (5 points)

```
51      if(waitpid(childPid,&status_new,0)==-1){ ... }
```

51 번째 line 의 waitpid(childPid, & status\_new, 0) code 에 의해, status\_new 는 child Process 의 종료 상태를 저장하게 됩니다. 58~66 번째 Line 들은 Child Process 의 상태를 확인합니다.

```
58      if(WIFEXITED(status_new)){
59          printf("exited, satus=%d\n",WEXITSTATUS(status_new));
60      }else if(WIFSIGNALED(status_new)){
61          printf("killed by signal %d\n",WTERMSIG(status_new));
62      }else if(WIFSTOPPED(status_new)){
63          printf("stopped by signal %d \n",WSTOPSIG(status_new));
64      }else if(WIFCONTINUED(status_new)){
65          printf("continued\n");
66      }
```

#### 1. WIFEXITED(status\_new)

Child Process 가 정상적으로 종료되었다면 True 를 반환합니다. 만약, 정상적으로 종료가 되어, 59 번째 Line 이 실행된다면, WEXITSTATUS(status\_new)가 실행되는데, 이 함수는 child process 의 exit status 를 반환합니다. 정확하게는, Status argument 의 least significant 8bits 를 반환합니다. 이 함수는 WIFEXITED(status\_new)의 반환값이 True 일 때만 사용해야합니다.

#### 2. WIFSIGNALED(status\_new)

Child Process 가 signal 에 의해 종료되었을 때, True 를 반환합니다. WTERMSIG(status\_new)는 child process 가 종료되도록 한 signal 의 번호를 반환합니다. 이 함수는 WIFSIGNALED(status\_new)가 True 를 반환했을 때만 호출되어야 합니다.

### 3. WIFSTOPPED(status\_new)

Child Process 가 signal 에 의해서 멈춰졌다면, true 를 반환합니다. WTOPSIG(status\_new)는 child process 를 멈추게 만든 signal 의 번호를 반환합니다. 이 함수는 WIFSTOPPED(status\_new)가 True 를 반환했을 때만 호출되어야 합니다.

### 4. WIFCONTINUED(status\_new)

멈춰졌던 Child process 가 SIGCONT signal 을 전달받아서, Child Process 가 재개되었을 때 True 를 반환합니다.

[Figure 5] is a part of myshellcommandsig.c file. Answer the following questions.

```
70     savedErrono = errno;
71
72     sigprocmask(SIG_SETMASK, &origMask, NULL);
73
74     sigaction(SIGINT, &saOrigInt, NULL);
75     sigaction(SIGQUIT, &saOrigQuit, NULL);
76
77     errno = savedErrono;
78
79     return status_new;
80 }
```

[그림 5 line 70 to line 80 of myshellcommandsig.c

6) Explain what lines 72 through 75 mean in Figure 3. (10 points)

```
51     if(waitpid(childPid,&status_new,0)==-1){
52         if(errno!=EINTR){
53             status_new=-1;
54             break;
55         }
56     }
```

70 부터 79 까지의 Line 들은 54 Line 의 switch 문 break 가 실행되었을 때에만 실행이 됩니다. Break 가 실행되기 위해서는 우선, waitpid 의 반환값이 -1 이어야 합니다.

Waitpid 의 반환값이 -1 인 의미는 해당 Process 가 임의의 자식 프로세스를 기다리고 있다는 의미 입니다. 또한 errno 이 EINTR 인 경우는 시스템 콜 수행 중 인터럽트가 걸려 수행이 중단된 경우입니다. 52Line 에서는 errno 이 EINTR 이 아닌 경우에만, 즉 시스템 콜 수행 중 인터럽트가 걸려 수행이 중단된 경우가 아닌 때에만 if 문이 true 를 반환합니다.

72, 74,75 Line 에서 사용되는 origMask, saOrigInt, saOrigQuit 의 값은 각각 19, 25, 26Line 에서 설정됩니다.

먼저, origMask 는 sigset\_t 변수로, 19Line 에서 다음과 같이 설정됩니다.

```
19 sigprocmask(SIG_BLOCK, &blockMask, & origMask);
```

기존에 해당 프로세스의 Block 된 Signal 들을 origMask Signal set 에 저장했습니다.

```
72 sigprocmask(SIG_SETMASK,&origMask,NULL);
```

그리고 72Line 에서 sigprocmask 함수를 사용하고, how 인자로 SIG\_SETMASK 를 설정하여 origMask 에 있는 signal 들을 해당 프로세스의 Blocked Signal 로 설정합니다.

saOrigInt와 saOrigQuit 은 sigaction 구조체 변수로, 각각 SIGINT와 SIGQUIT 에 대한 해당 process 의 기존의 action 을 저장합니다.

```
25 sigaction(SIGINT,&saIgnore,&saOrigInt);
```

```
26 sigaction(SIGQUIT,&saIgnore,&saOrigQuit);
```

그리고 74, 75Line 에서 sigaction 함수를 사용하여 각각 SIGINT 와 SIGQUIT 에 대한 action 을 초기에 설정된 action 으로 수정합니다.

```
74 sigaction(SIGINT,&saOrigInt,NULL);
```

```
75 sigaction(SIGQUIT,&saOrigQuit,NULL);
```

따라서, 72~75Line 을 실행하면, Blocked Signal 들은 shellcommand 함수 실행 초반(19~26Line)의 Blocked signal 들로 설정 되고, SIGINT 와 SIGQUIT 에 대한 action 들 또한, shellcommand 함수 실행 초반의 action 으로 설정될 것입니다.

[Figure 6] is a part of myshellcommandsig.c file. Answer the following questions.

```
82 int main(int argc, char *argv[])
83 {
84     char str[MAX_CMD_LEN]; // for user typed command with maximum command length
85     int status; // for the return value of shellcommand()
86
87     for (;;) {
88         printf("Command: ");
89         fflush(stdout); // for fflushing standard output
90
91         if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
92             break;
93
94         status = shellcommand(str); // for user's shell command
95
96         printf("shellcommand() returned: status = %d\n", (unsigned int) status);
97     }
98     exit(EXIT_SUCCESS);
99 }
100 }
```

[그림 6] line 82 to line 100 of myshellcommandsig.c

7) Explain what lines 87 through 97 mean in Figure 6. (10 points)

87Line 의 for 문은 while(1)과 같은 의미를 갖습니다. 즉, for 문을 break 할 조건문이 없으므로, for 문안의 code 들은 계속 반복됩니다.

88~89Line

```
88     printf("Command:");
89     fflush(stdout);
```

실행창에 Command: 를 출력하고, standard output steam 를 flush 합니다.

```
Command: 
```

91~92Line

```
90
91     if(fgets(str,MAX_CMD_LEN,stdin)==NULL)
92         break;
```

fgets 함수를 사용하여 최대 MAX\_CMD\_LEN 개의 문자열을 stdin(standard input)을 통해서 입력 받습니다. Standard input 은 키보드 이고, fgets 를 통해 입력 받은 문자열을 str char array 에 저장합니다.

94 Line

```
94      status=shellcommand(str);
```

입력 받은 문자열을 `shellcommand` 함수에 전달합니다. 그리고, `shellcommand` 함수가 반환한 값을 `status int` 형 변수에 저장합니다.

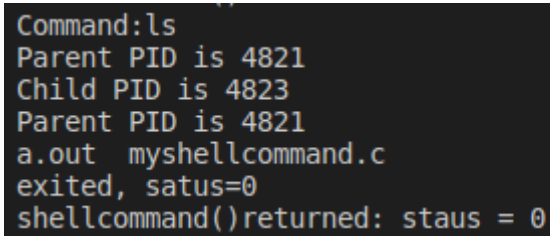
96 Line

```
96      printf("shellcommand()returned: staus = %d\n",(unsigned int )status);
```

94Line 에서 저장한 `status` 를 이용하여 문자열을 출력합니다.

88~97Line 이 1. 정상적인 `command`, 2. 비정상적인 `command` 에 대해서 어떻게 반응하는지 결과를 정리하겠습니다.

#### 1. 정상적인 `command`



```
Command:ls
Parent PID is 4821
Child PID is 4823
Parent PID is 4821
a.out myshellcommand.c
exited, satus=0
shellcommand()returned: staus = 0
```

[그림 7] 정상적인 Command 를 입력했을 때 실행 화면

Shellcommand 의 28Line 에서 `fork` 를 사용하여 `child process` 를 생성합니다.

```
28      switch(childPid=fork()){
```

28Line 으로 인해 `childPid` 에는 생성한 `child process` 의 `PID` 가 저장됩니다. 이때, `child Process` 가 정상적으로 생성되었다면, `ChildPid` 에는 0 보다 큰 값이 저장되어, `switch` 의 `case block` 중, `default block` 이 실행이 됩니다. 이 `Default block` 을 실행하는 `Process` 는 `Parent Process` 입니다.

```
51          default:
52              printf("Parent PID is %ld\n",(long)getpid());
53              if(waitpid(childPid,&status_new,0)==-1){
```

그리고, 52 Line 의 `Printf` 함수를 호출하면서 `parent process` 인 자신의 `PID` 를 출력하고, `waitpid` 함수를 호출하여, 자신은 `wait` 상태가 되고, `Child Process` 가 실행됩니다.

`Child process` 는 31Line 의 `case 0: block` 부터 실행합니다.

44,45 Line 에서 자신의 PID 와 Parent Process 의 PID 를 출력하고, 47 Line 에서 execl 를 호출하여 shell program 을 자신에 메모리에 덮어써서 실행합니다.

```
44     printf("Child PID is %ld\n",(long)getpid());
45     printf("Parent PID is %ld\n",(long)getppid());

47     execl("/bin/sh","sh","-c",command,(char*)NULL);
```

정상적인 Command 의 경우, shell program process 가 정상적으로 종료가 되어서, 53Line 의 status\_new int 변수에는 0 이 저장됩니다.

따라서, 60Line 의 WEXITSTATUS(status\_new)는 status new 변수의 하위 8 비트인 0 을 반환하게 됩니다.

```
59     if(WIFEXITED(status_new)){
60         printf("exited, satus=%d\n",WEXITSTATUS(status_new));
```

68Line 의 return 문에 의해서 shellcommand 함수는 status\_new 을 반환합니다. Status\_new int 변수에는 0 이 저장되었으므로, 0 을 반환합니다.

```
68     return status_new;
```

따라서, Main 함수의 97Line 에서 출력하는 status 의 값은 0 이 됩니다.

```
97     printf("shellcommand()returned: staus = %d\n",(unsigned int )status);
```

## 2. 비정상적인 command

```
Command:wrong command
Parent PID is 5076
Child PID is 5079
Parent PID is 5076
sh: 1: wrong: not found
exited, satus=127
shellcommand()returned: staus = 32512
```

[그림 8] 비정상적인 Command 를 입력했을 때 실행화면

잘못된 command 가 입력되었을 때, 53 line 으로 인해 status\_new 에 저장되는 값이 달라집니다.

```
53     if(waitpid(childPid,&status_new,0)==-1){
```

잘못된 command 가 입력되었을 때에는 child process 가 정상적으로 종료되지 않아서 status\_new 에는 0 이 아닌 값이 저장됩니다. status\_new 에 32512 가 저장되고,



제목 <b>AWS</b> 인스턴스 및 원격접속 결과보고서	소제목 과제물 <b>1</b>
------------------------------------	---------------------

32512 의 하위 8 비트의 값은 127 입니다. 따라서, WEXITSTATUS(status\_new) 매크로는 127 을 반환하고, shellcommand 함수는 32512 를 반환합니다.