# OS Assignment#1

**리눅스의 프로세스 자동 관리 프로그램 추가 기능 구현**

**학부 : 소프트웨어 학부**

**학번 : 20150291**

**이름 : 하현수**

# 1. 소개

## 과제 개요

본 과제는 리눅스 운영체제에서 프로세스 실행 / 종료를 자동으로 관리하는 프로그램을 구현하는 것을 목표로 합니다.
기본 사항으로 주어진 procman.c 라는 프로세스 관리 프로그램의 소스코드를 수정하여 sigaction() 대신 signalfd()를 이용하여 SIGCHLD 시그널을 처리하고 프로그램 실행 정보에 order항목을 성공적으로 추가하였습니다.
구현에 대한 테스트는 Vmware Workstation 12 버전을 통한 우분투 16.04 가상머신을 통해 진행하였습니다.
관련 연구의 소스 분석 부분에서는, 수정 전의 원본 코드에 대한 분석을 한 뒤, 추가 기능 구현 방법 부분에서 수정한 소스 코드를 설명하도록 하겠습니다.

## Config 파일 수정 사항

기본적으로 주어진 config1.txt 파일에 여러 가지 수정 사항이 있습니다.

```
# normal tasks
notcmd:once:101::./TTT
id1:once:1::./task -n Task1 -w Hi -r -t 1
id2:once:5:id1:./task -n Task2 -w Hello -r -t 1
idwait1:wait:28::./task -n Task3 -t 2
respawn1 : respawn :8: : ./task -n Task4 -t 4
idonce1:once:101::./task -n Task5 -t 1
idonce2:once:153::./task -n Task6 -t 2
```

Normal Tasks의 각각의 항목에 order 값을 임의로 추가하였습니다.
idonce1과 notcmd의 order값을 모두 101로 입력하여, 동일 order 처리가 입력 순으로 이루어지는지를 확인 할 수 있도록 하였습니다.

id2의 액션이 wait일 경우 task 정보가 제대로 등록되지 않게 되어 id1파이프 정보가 등록되지 않아 아래 예외 사항 id11이 정상적으로 등록되는 문제가 있어

```
# already piped tasks are not allowed to pipe
id11:once::id1:./task -n Task11
id12:once::id2:./task -n Task12
```

id2에 해당하는 라인의 경우 wait예외처리는 idwait1에서 확인이 가능하기 때문에, id2의 action은 once로 바꿔 정상적으로 동작하게 하였습니다.

```
# invalid action
id4:restart:::

# invalid order
id3:once:10000::./task -n Task1 -w Hi -r -t 1
id4:once:abc::./task -n Task1 -w Hi -r -t 1

# empty task
id5:once:::
```

Order가 4자리 숫자를 넘어 갔을 때의 예외처리를 나타내기 위하여, invalid action과 empty task 사이에 invalid order에 대한 예외 task를 추가하였습니다.

# 2. 관련 연구

## 소스 분석

소스 분석은 소스코드가 너무 긴 관계로, procman.c의 각 함수를 Visual Studio의 함수 접기 기능을 통해서 원형을 보고 설명하겠습니다. Task.c는 수정 사항과 분석할 내용이 크게 없어 분석하지 않았습니다.

```
typedef struct _Task Task;
struct _Task
{
  Task          *next;

  volatile pid_t pid;
  int            piped;
  int            pipe_a[2];
  int            pipe_b[2];

  char           id[ID_MAX + 1];
  char           pipe_id[ID_MAX + 1];
  Action         action;
  char           command[COMMAND_LEN];
};
```

**Task 구조체**는 프로세스 들의 정보를 가지고 있는 링크드리스트의 정의부입니다.
본 구조체는 다음 노드의 포인터를 가진 next와 프로세스 id인 pid에 대한 정보와 pipe 정보, 그리고 자체적인 식별 id값과 action의 값
(본 과제에서는 once 혹은 respawn의 경우 두가지)
그리고 마지막으로 Command에 대한 정보를 가지고 있습니다.
Task 링크드리스트는 read_config 함수에서 정보를 받아오게 됩니다.
그 내용은 아래서 다시 분석하겠습니다.

```
static Task *tasks;

static volatile int running;
```

Task 링크드리스트의 Head 역할을 하는 **tasks**라는 포인터를 전역 변수로 선언하였고, 실행 여부를 표시하는 running 변수도 선언되어 있습니다.

```
static char *
strstrip (char *str) { ... }
```

**strstrip 함수**는 문자열의 공백 부분에 널 문자를 채우는 역할과 함께 read_config에서 :부분에 널값을 채우고 strstrip함수를 이용해서 해당 문자열에서 필요한 값만 파싱합니다. 자세한 사항은 read_config부분에서 설명하겠습니다.

```
static int
check_valid_id (const char *str) { ... }
```

**check_valid_id 함수**는 id에 해당하는 string을 받아 온 뒤, ID의 최소 길이와 최대길이를 비교하고, 소문자와 숫자외에 다른 글자가 있는지 확인하여 ID의 유효성을 체크하는 함수입니다. read_config의 예외처리에 사용됩니다.

```
static Task *
lookup_task (const char *id) { ... }
```

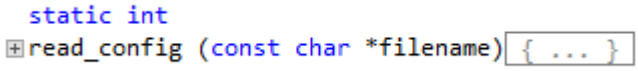
**lookup_task 함수**는 id값을 이용하여 등록된 task 링크드리스트에 입력된 id와 같은 id값을 가진 task가 있는지 검사하는 함수입니다.
read_config의 중복 id 예외처리에 사용됩니다.

```
static Task *
lookup_task_by_pid (pid_t pid) { ... }
```

**lookup_task_by_pid함수**는 pid를 입력 받아 해당 pid에 해당하는 Task를 반환하는 함수입니다. **wait_for_children함수**에서 pid를 입력 받아 task를 받아 올 때 사용됩니다.

```
static void
append_task (Task *task) { ... }
```

**append_task함수**는 링크드리스트에 task를 삽입하는 역할을 합니다.
링크드리스트가 비어있다면, head에 해당하는 **tasks 전역 포인터 변수**에 new_task를 삽입하고, 만약 head가 존재한다면 링크드리스트의 맨 뒤 부분에 삽입됩니다. 이 부분을 수정하여 오름 차순 정렬 링크드리스트를 만들면 Order기능을 구현할 수 있습니다.



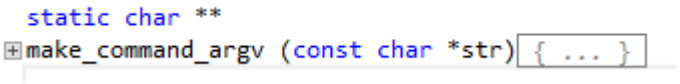

**read_config함수**는 config txt파일에서 정보를 읽어오며, Task 링크드리스트에 정보를 불러오는 역할을 합니다.

id:action:pipe-id:command 형식으로 이루어진 config 텍스트 파일을 한줄씩 읽어 내려가는 루프문으로 구성되어 있으며, 각 항목 분류에 따라서 :를 널로 변경시킨 뒤, **strstrip함수**를 이용하여 : 사이의 값을 파싱받아 Task 노드에 값을 입력합니다.

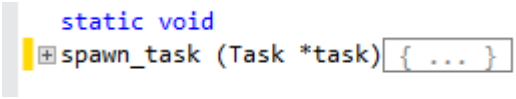
**check_valid_id 함수**나 **lookup_task 함수**등을 통해서 유효성을 검증하며,

만약 유효하지 않은 값이 입력된 경우, goto 문을 통해 루프의 맨 끝 부분으로 보내고 Invalid format이라는 오류 메시지를 출력합니다.

action, pipe-id, command에 대한 유효성 검증과 정보 입력이 완료되면 **append_task 함수**를 통해 링크드리스트에 입력된 정보가 삽입됩니다.




**make_command_argv 함수**는 command 부분에 있는 명령어의 유효성을 검증하고, 공백(띄어쓰기) 단위로 command를 나눠서 argv라는 2차원 char형 배열에 저장하는 역할을 합니다.




**spawn_task 함수**는 해당하는 task를 실행시키기 위한 함수입니다. **pipe시스템콜**을 이용하여 예외처리를 하며, **fork 시스템콜**을 통해서 프로세스를 복제하고, 파이프 작업을 **dub2 시스템콜**을 이용하여 마친 후에, **make_command_argv함수**를 이용하여 받아온 argv라는 2차원 배열을 입력 받는 **execvp 시스템콜**를 이용하여 command를 실행시킵니다.

```
static void
spawn_tasks (void) { ... }
```

**spawn_tasks 함**수는 링크드리스트를 탐색하며, **spawn_task 함수**를 호출합니다. 링크드리스트를 헤드부터 읽기 때문에, order값 순서대로 정렬된 링크드리스트라면 차례대로 실행되며 Order의 순서를 따를 수 있겠다고 추론하였습니다.

```
static void
wait_for_children (int signo) { ... }
```

**wait_for_children 함수**는 **waitpid 시스템콜**을 통해서 자식 프로세스들의 신호를 받는다. 만약 해당 Task의 action이 Respawn이고 running 이 1이라면,
해당 Task를 **spawn_task**하고 만약 그렇지 않다면 task의 pid를 0으로 마친다.
task의 pid가 0이 되지 않는 한, goto문에 의해서 무한루프가 반복된다.

```
static void
terminate_children (int signo) { ... }
```

**terminate_children 함수**는 running 상태를 0으로 만들고 모든 프로세스를 **kill 시스템콜**을 이용하여 종료시킨다.

```
int
main (int    argc,
      char **argv) { ... }
```

**main 함수**는 시작 인수로 config 파일의 파일명을 받아 **read_config함수**를 실행시킨다. 정상적으로 config 파일을 로드했다면, running 값을 1로 바꾼 뒤 시그널을 입력받는다. **SIGCHLD 시그널**을 받았을 경우 **sigaction 시스템콜**을 이용하여 **wait_for_children 함수**를 핸들러 함수로 사용하고,
**SIGINT와 SIGTERM시그널**을 받았을 경우 **terminate_children 함수**를 핸들러 함수로 이용한다.
또한 마지막 while문에서는 모든 프로세스가 종료될 때까지 루프를 반복하며,
모든 프로세스의 pid가 0보다 작아졌을 경우, 정상적으로 프로그램이 종료되게 된다.

# 사용된 시스템 콜 / API 함수 연구

해당 과제에 이용된 다소 사소할 수 있는 모든 함수와 시스템 콜을 모두 적었습니다, 시스템 콜과 Library Call을 비교하는 것이 흥미로웠습니다.

**memmove() 함수**: memcpy() 처럼 메모리 영역을 복사하는 함수입니다. 다른 점은 하나의 포인터에 대해서 동일한 영역 내에서 복사가 가능합니다. string.h 헤더파일에 존재합니다.

**isspace() 함수** : 문자가 공백 문자인지 판별하는 함수입니다. 헤더파일은 ctype안에 있으며, 공백이 아닐 경우 0을 리턴합니다.

**islower() 함수** : 문자가 소문자인지 판별합니다. 헤더파일은 ctype.h 이며, 소문자가 아닐 경우 0을 리턴합니다.

**isdigit() 함수** : 문자가 숫자인지 판별합니다. 헤더파일은 ctype.h 이며, 숫자가 아닐 경우 0을 리턴합니다.

**strcmp() 함수** : 문자열을 비교합니다. 헤더파일은 string.h 이며, 같을 경우 0를 리턴합니다.

**malloc() 함수** : 메모리 할당은 OS 고유의 기능이지만, malloc 함수는 시스템콜이 아닌 내부에 메모리를 할당하여 return하는 library call방식의 함수입니다. 헤더파일은 stdlib.h이며, 메모리가 할당된 포인터를 리턴합니다.

**fopen() 함수** : open() 시스템콜과 달리 내부적으로 라이브러리를 이용한 Library Call 함수입니다. 파일을 열고 해당 파일 포인터를 리턴합니다. 헤더파일은 stdio.h 입니다.

**memset() 함수** : 할당 받은 메모리를 특정 값으로 초기화 하는 함수입니다.
메모리 관련 함수 이지만 헤더 파일은 string.h 파일입니다.

**strchr() 함수** : 문자열에서 찾는 문자에 해당하는 포인터 값을 리턴하는 함수 입니다. 헤더파일은 string.h 입니다.

**strcpy() 함수** : 문자열을 복사하는 함수입니다. 복사가 완료된 포인터 값을 리턴하며 헤더파일은 string.h 입니다.

**strcasecmp() 함수** : 대소문자를 무시하고 문자열을 비교하는 함수입니다. 같을 경우 0을 리턴합니다. 헤더파일은 string.h 입니다.

**strncpy() 함수** : 문자열을 지정한 길이 만큼 복사하는 함수입니다. 헤더 파일은 string.h 이며 복사한 문자열을 반환합니다.

**fclose() 함수** : fopen으로 열은 파일을 닫는 파일 포인터입니다. 헤더파일은 stdio.h이며, 정상적으로 닫았다면 0을, 오류가 발생하면 EOF를 반환합니다. close() 시스템콜과 다른, 라이브러리 콜 함수입니다.

**calloc() 함수** : malloc() 처럼 메모리를 동적으로 할당 받습니다. Malloc과의 차이점은 바이트 단위 메모리 요청이 아닌 특정 크기에 대한 개수 만큼을 메모리로 요청할 수 있다는 점이며, 헤더파일은 stdlib.h입니다.

**strdup() 함수** : 복사할 문자열 크기에 맞는 메모리를 확보한 후 문자열을 복사한 뒤 확보한 메모리의 포인터를 반환하는 함수입니다. 헤더파일은 string.h 입니다.

**pipe() 시스템콜** : 프로세스끼리의 통신(IPC)에서 사용하는 파이프를 생성합니다. pipe()에서 생성하는 파이프는 프로세스 안에 생성되는 것이 아니라 커널에 생성되며 프로세스에는 다만 파이프를 이용할 수 있는 파일 디스크립터만 제공됩니다. 그러므로 하나의 프로세스에서 파이프를 생성했다고 해도 다른 프로세스에서 그 디스크립터를 사용할 수 있다면, 그 디스크립터를 이용하여 서로 통신할 수 있습니다. 그러나 pipe()에서 생성한 파이프의 문제점은 입출력 방향이 결정되어 있다는 점입니다. pipe()를 실행하면 파이프를 이용할 수 있는 2개의 파일 디스크립터가 구해집니다. int pipe(int filedes[2]); 에서, 2개의 파일 디스크립터중 filedes[0] 은 파이프의 읽기 전용 디스크립터, filedes[1]은 쓰기 전용 디스크립터입니다. 헤더파일은 unistd.h 입니다.

**fork() 시스템콜** : fork() 시스템콜은 현재 실행되는 프로세스에 대해 복사본 프로세스를 생성합니다. fork() 시스템콜은 하나의 프로그램을 통째로 복제하며, 부모에게는 복제된 자식의 pid값이 반환되며, 복제된 프로세스는 반환되는 pid값이 0이라는 점만 다를 뿐입니다. 헤더파일은 unistd.h 입니다.

**exit() 함수** : exit()는 프로세스를 종료하는 시스템콜이기도 하지만, 본 과제에서 사용된 exit(-1) 등은 stdlib.h 에 속한 c언어 라이브러리 함수라는 것을 visual studio 2015버전을 통하여 확인하였습니다. 반환값은 main() 함수의 반환값에 해당하는 정수 입니다.

**dup2() 시스템 콜** : dup2()는 dup()과 달리 프로그래머가 원하는 번호의 디스크립터를 받을 수 있습니다. 프로그래머가 지정하는 번호가 이미 사용하는 디스크립터 번호더라도, dup2 시스템콜은 자동으로 그 파일을 닫고 다시 지정해 줍니다.
헤더 파일은 unistd.h 이며, 형태는 int dup2(int fildes, int fildes2); 입니다.
fildes는 파일 디스크립터, fildes2는 원하는 파일 디스크립터 번호입니다.
원하는 파일 디스크립터 번호가 리턴되며, 실패 시 -1이 리턴됩니다.
**close() 시스템 콜** : 열기를 한 파일 사용을 중지합니다. Open() 시스템 콜은 fcntl.h에 정의되어 있지만 close() 시스템콜은 unistd.h 에 정의되어 있습니다.
정상적으로 close했다면 0을 실패했다면 -1이 리턴됩니다.
**execvp() 시스템 콜** : exec기반의 다른 프로그램 실행 시스템 콜입니다. Exec 함수는 다른 프로그램을 실행하고 자신은 종료하는 특성이 있으며, 명령 라인 인수로 인수 배열을 받고, 환경 설정이 불가능 합니다. 헤더파일은 unistd.h입니다.
**waitpid() 시스템 콜** : wait() 시스템 콜 처럼 자식 프로세스가 종료될 때 까지 대기합니다. 차이점은 wait() 시스템콜은 자식 프로세스 중 어느 하나라도 종료되면 복귀되지만, waitpid 시스템콜은 특정 자식 프로세스가 종료될 때까지 대기합니다.
또한 waitpid는 WNOHANG옵션을 사용하면 block되지 않고 다른 작업을 진행할 수 있습니다. 첫번째 인수 pid_t pid는 감시 대상인 자식 프로세스의 ID입니다.
하지만 pid뿐만 아니라, 0을 넣을 경우 현재의 프로세스 그룹 ID와 같은 그룹의 자식 프로세스가 종료되면 복귀, -1을 넣을 경우 wait()와 같이 아무 자식이라도 종료되면 복귀등의 다양한 값을 지정할 수 있습니다.
3번째 인수 option에는 가장 많이 넣는 값으로는 WNOHANG과 0을 뽑을 수 있는데, WNOHANG은 부모 프로세스의 block을 막아주는 역할, 0은 wait()와 같이 자식 프로세스가 종료될 때 까지 부모가 block되도록 하는 것을 의미합니다.
종료된 자식 프로세스의 pid를 반환하며, 실패시 -1, WNOHANG을 이용하고 자식 프로세스가 종료되지 않았다면 0이 리턴됩니다. 헤더 파일은 unistd.h 입니다.

**kill() 시스템 콜** : 쉘에서 프로세스를 죽이는 kill 명령과는 달리, 프로세스에 시그널을 전송하는 시스템콜입니다. 프로세스에 SIGKILL을 보내면 쉘 명령의 kill과 같은 역할을 합니다. Kill은 특정 프로세스 뿐만 아니라 그룹 id가 같은 모든 프로세스에게 동시에 시그널을 전송할 수 있으며, 권한 안에 있는 모든 프로세스에게 시그널을 전송합니다.

원형은 int kill(pid_t pid, int sig); 입니다.

pid에 양의 정수를 넣을 경우 지정한 프로세스의 pid로만 시그널을 전송하며,

0을 넣을 경우, 함수를 호출하는 프로세스와 같은 그룹에 있는 모든 프로세스에 시그널을 전송하며,

-1을 넣을 경우, 함수를 호출하는 프로세스가 전송할 수 있는 권한을 가진 모든 프로세스에 시그널을 전송합니다.

-1 외의 음수를 넣을 경우, 첫번째 인수 pid의 절대값 프로세스 그룹에 속하는 모든 프로세스에 시그널을 전송합니다. 헤더파일은 signal.h 입니다.

성공시 0, 실패시 -1이 리턴됩니다.

**sigaction() 시스템 콜** : sigaction 시스템 콜은 signal 시스템 콜보다 향상된 기능을 제공하는 시그널 처리 함수입니다. signal() 에서는 처리할 행동 정보로 시그널이 발생하면 호출이 될 함수 포인터를 넘겨주었으나, sigaction은 struct sigaction 구조체 값을 사용하기 때문에 좀 더 다양한 지정이 가능합니다.

```
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_handler 대신에 동작하는 핸들러
    sigset_t sa_mask;             // 시그널을 처리하는 동안 블록화할 시그널 집합의 마스크
    int sa_flags;                 // 아래 설명을 참고하세요.
    void (*sa_restorer)(void);    // 사용해서는 안됩니다.
}
```

ss_flgs는 아래와 같은 값이 사용되며, OR 연산자로 여러 개를 동시에 지정하여 사용할 수 있습니다.

| 옵션 | 의미 |
|---|---|
| SA_NOCLDSTOP | signum이 SIGCHILD일 경우, 자식 프로세스가 멈추었을 때, 부모 프로세스에 SIGCHILD가 전달되지 않는다. |
| SA_ONESHOT 또는 SA_RESETHAND | 시그널을 받으면 설정된 행도을 취하고 시스템 기본 설정인 SIG_DFL 로 재설정된다. |
| SA_RESTART | 시그널 처리에 의해 방해 받은 시스템 호출은 시그널 처리가 끝나면 재시작한다. |
| SA_NOMASK 또는 SA_NODEFER | 시그널을 처리하는 동안에 전달되는 시그널은 블록되지 않는다. |
| SA_SIGINFO | 이 옵션이 사용되면 sa_handler대신에 sa_sigaction이 동작되며, sa_handler 보다 더 다양한 인수를 받을 수 있습니다. sa_sigaction이 받는 인수에는 시그널 번호, 시그널이 만들어진 이유, 시그널을 받는 프로세스의 정보입니다. |

헤더파일은 signal.h 이며, 성공시 0 실패시 -1이 리턴됩니다.

**signalfd() 시스템콜** : 커널 2.6.22부터 이용할 수 있는 새로운 리눅스 특유의 시스템콜로, 파일 디스크립터를 사용해서 시그널을 받을 수 있습니다. 핸들러 함수는 제공하지 않지만, 동기 방식으로 시그널을 처리할 수 있습니다.

signalfd를 사용하기 위해서는 sigprocmask 시스템콜을 이용해서 처리할 시그널을 블록해야 합니다. Sigaction과 마찬가지로 구조체를 사용하며

struct signalfd_siginfo를 이용하여 정보를 넣습니다.

특징은 fork()후에도 파일 디스크립터는 닫히지 않으므로, 자식 프로세스가 부모 프로세스에 전달된 시그널을 읽을 수 있다는 점입니다.

헤더파일은 sys/signalfd.h 입니다.

# 3. 추가 기능 구현 방법

## Signalfd를 이용한 SIGCHLD 처리법

```
#include <sys/signalfd.h>//signalfd를 사용하기 위한 헤더 파일
#define MSG(x...) fprintf (stderr, x)
#define STRERROR  strerror (errno)
```

signalfd 시스템콜을 사용하기 위해서 sys/signalfd.h 헤더를 인클루드합니다.

```
int
main(int    argc,
    char **argv)
{//============================================Edit Part=============================================
    struct sigaction sa;
    struct signalfd_siginfo fdsi;//fdsi라는 signalfd용 구조체를 생성합니다.
    int terminated;
    srand((unsigned int)time(NULL));//rand()를 위한 시간 씨드 생성합니다.
    //==================================================================================================
```

메인함수에 윗부분에 signalfd_siginfo형식의 구조체 fdsi를 선언합니다.

```
/* SIGCHLD */
/*sigemptyset (&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = wait_for_children;
if (sigaction (SIGCHLD, &sa, NULL))
  MSG ("failed to register signal handler for SIGINT\n");*/
//=========================================Edit Part=============================================
sigset_t mymask;
int mysfd;
ssize_t s;//signalfd에 필요한 변수들입니다.

sigemptyset(&mymask);//mymask를 비웁니다.
sigaddset(&mymask, SIGCHLD);//SIGCHLD 시그널을 처리하기 위해 추가하였습니다.
sigaddset(&mymask, SIGQUIT);//SIGQUIT 시그널을 처리하기 위해 추가하였습니다.
mysfd = signalfd(-1, &mymask, 0);//signalfd에 mask값을 입력합니다.
if (mysfd == -1)//-1이 반환될 경우 오류메세지를 띄워줍니다.
    MSG("failed to signal handler SIGCHLD");
if (sigprocmask(SIG_BLOCK, &mymask, NULL) == -1)//sigprocmask 시스템콜을 호출합니다.
{
    MSG("error");
}//==================================================================================================
/* SIGINT */
```

기존 sigaction을 이용하는 SIGCHLD 처리 부분을 주석처리 한뒤, mymask과 mysfd와 s등의 필요 변수를 선언하였습니다.

mymask를 sigemptyset(&mymask)를 이용하여 비워준뒤, sigaddset(&mymask,SIGCHLD)와 sigaddset(&mymask,SIGQUIT)을 추가하여 처리할 시그널을 mymask에 추가해 줍니다.

그 후 signalfd(-1, &mymask, 0);을 호출한뒤, mysfd가 -1일 경우 오류 메시지를

띄워줍니다.

그후 sigprocmask 시스템콜을 호출하여, 시그널이 발생하더라도 대기상태로 만들어줍니다.

```
terminated = 0;
while (!terminated)
{
    Task *task;

    terminated = 1;
    for (task = tasks; task != NULL; task = task->next)
        if (task->pid > 0)
        {
            terminated = 0;
            break;
        }
    //==============================Edit Part==========================================================
    s = read(mysfd, &fdsi, sizeof(struct signalfd_siginfo));//signalfd의 구조체를 읽어옵니다.
    if (s != sizeof(struct signalfd_siginfo))//제대로 불러오지 못했을 경우 오류를 출력합니다.
        MSG("read error₩n");
    if (fdsi.ssi_signo == SIGCHLD)//만약 입력받은 시그널이 SIGCHLD일 경우
    {
        wait_for_children(SIGCHLD);//동기방식으로 wait_for_children함수를 호출합니다.
        //printf("Got SIGCHLD₩n");//시그널을 제대로 받아오는지 체크하기 위한 출력부입니다.
    }
    else if (fdsi.ssi_signo == SIGQUIT)//SIGQUIT 시그널이 올경우
    {
        exit(EXIT_SUCCESS);//종료시킵니다.
    }
    else {
        printf("Read Unexpected signal₩n");//전혀 이상한 시그널을 받았을때의 예외처리입니다.
    }

    usleep(100000);
}
//=================================================================================================
return 0;
```

그후 main문 하단에 있는 while 무한 루프 문장에 signalfd의 구조체를 읽어오는 read시스템콜 코드를 추가합니다.

그 후, 만약 signalfd의 구조체 fdsi의 signo(시그널)이 SIGCHLD일 경우 wait_for_children 함수를 호출하여 작동시킵니다.

이 과정을 확인하기 위해 printf문을 추가하였습니다.

다음은 SIGCHLD를 잘 받아오는 모습을 캡처 한 것입니다.

```
Got SIGCHLD
'Task4' start (timeout 4)
'Task4' end
Got SIGCHLD
'Task4' start (timeout 4)
'Task4' end
Got SIGCHLD
'Task4' start (timeout 4)
'Task4' end
Got SIGCHLD
'Task4' start (timeout 4)
'Task4' end
```

Task4를 리스폰하는 명령문을 받으며, 정상적으로 SIGCHLD 코드를 받아오고 있으며, 계속해서 Task4를 Start하는 모습을 볼 수 있습니다.

# Config파일에 Order추가하고 Order순서로 실행

```
#include <time.h>//srand함수를 사용하기 위한 헤더 파일
#include <sys/wait.h>
```

오더가 공백일 경우 임의로 섞어 실행하기 위한 rand함수를 사용하기위한 time.h 헤더파일을 추가하였습니다.

```
typedef struct _Task Task;
struct _Task
{
    Task          *next;

    volatile pid_t pid;
    int            piped;
    int            pipe_a[2];
    int            pipe_b[2];
    //====================================Edit Part=================================================
    int            order;//order항목 추가
    //==============================================================================================
    char           id[ID_MAX + 1];
    char           pipe_id[ID_MAX + 1];
    Action         action;
    char           command[COMMAND_LEN];
};
```

Task 노드에 order에 대한 정보가 추가되어야 하므로, 링크드리스트 구조체에 order라는 int형 항목을 추가하였습니다.

```
}
//====================================Edit Part=================================================
static int
check_valid_order(const char *str)//order가 4자리를 넘어가거나, 숫자가 아닐 경우 예외처리를 위한 함수
{
    size_t len;
    int    i;

    len = strlen(str);//문자열의 길이 측정
    if (len > 4)//만약 4자리수를 넘어 갈경우
        return -1;//-1반환하여 유효하지 않음을 반환

    for (i = 0; i < len; i++)//4자리수 이내의 경우일지라도
        if (str[i] < '0' || str[i] > '9')//0~9사이의 숫자인지 확인함
            return -1;//숫자가 아닌 문자가 들어있을 경우 유효하지 않음을 반환

    return 0;//아무 문제 없을시 0 반환
}//==============================================================================================
```

order에 들어가는 값이 유효한지 검증하는 check_valid_order함수를 구현하였습니다. 기본적인 뼈대는 check_valid_id 함수와 비슷하며, 명세서에 Order가 4자리수 이내여야 한다고 적혀있으므로, 4자리수를 넘어가는 문자열은 우선 -1을 반환하도록 구현하였습니다. 또한 4자리수를 넘지 않더라도, 숫자가 아닌 문자가 들어있을 경우에도 유효하지 않음을 의미하는 -1을 리턴하도록 구현하였습니다.

```
new_task->next = NULL;
//=====================================Edit Part=============================================
if (!tasks)
    tasks = new_task;
else
{
    Task *t;
    if (new_task->order < tasks->order)//new_task의 오더가 head의 오더보다 작을경우 head 앞에 붙임
    {
        new_task->next = tasks;//new_task 다음 노드를 tasks로 설정
        tasks = new_task;//head를 new_task로 변경
    }
    else if (new_task->order >= tasks->order)//만약 헤드의 order값이 new_task의 order값보다 클경우
    {
        t = tasks;
        while (t != NULL && new_task->order >= t->order)//자신보다 최초로 같거나 큰order값을 찾은 후 해당
        {                                               //노드의 바로 앞에 삽입함
            temp = t;
            t = t->next;
        }
        new_task->next = temp->next;//링크드리스트 재연결
        temp->next = new_task;//연결 완료
    }
}//append_task를 변경하여 자동으로 order순으로 오름차순 정렬되는 링크드리스트를 만들었습니다.
=============================================================================================
```

그 후 append_task 함수에서 링크드리스트의 삽입 방식을 변경하였습니다.

order를 기준으로 오름차순으로 삽입되는 링크드리스트를 만들어서 배치해야 spawn_tasks를 호출할 때 순서대로 실행이 가능하기 때문에, 오름차순으로 삽입되는 링크드리스트를 구현하였습니다.

만약 삽입하고자하는 노드의 order값이 head 노드인 tasks의 order값보다 작을 경우에는, 맨 앞에 삽입하고, head를 new_task로 바꿉니다.

혹은 삽입하고자하는 노드 new_task의 order값보다 head 노드인 tasks의 order값이 더 작을 경우, for문을 통해 링크드리스트를 탐색하며, 최초로 new_task의 order보다 크거나 같은 order값을 가진 노드를 찾은 뒤, 바로 그 앞에 삽입합니다.

이런식으로 append_task함수를 변경하게 되면, 링크드리스트는 항상 오름차순을 유지하게 되고, order순으로 정렬된 링크드리스트를 항상 유지할 수 있습니다.

```
,
//==================================Edit Part=================================================
/* Order */
s = p + 1;//s = p + 1 을 통해서 다음 문자열 위치로 넘겨 줍니다.
p = strchr(s, ':');// :에 해당하는 문자열 포인터를 찾아 p에 입력합니다.
if (!p)//만약 포맷이 잘못됬을 경우 invalid_line 분기로 goto 시킵니다.
{
    goto invalid_line;
}
*p = '₩0';// :에 해당하는 부분을 널 문자로 바꿔줍니다.
strstrip(s);//:가 널문자로 변경되어 strstrip을 통한 파싱이 가능합니다.
if (check_valid_order(s))//최종적으로 나온 order 문자열을 검사하는 함수입니다.
{
    MSG("invalid order '%s' in line %d, ignored₩n", s, line_nr);//유효하지 않은 값일경우 오류 메세지 출력
    continue;
}
if (strlen(s) == 0)//공백일 경우 임의의 순서를 정합니다.
    task.order = rand() % 100 + 10000;//오더는 무조건 0~9999사이의 숫자이기 때문에 rand() % 100 + 10000을
                                      //이용하면 order가 공백이 아닌 task보다는 순서가 무조건 뒤로 가되,
else                                  //임의의 순서로 실행되도록 구현할 수 있습니다.
    task.order = atoi(s);//공백도 아니고 유효성 검사도 통과했다면 문자열을 숫자로 변환하여 저장
//============================================================================================
```

그 다음에는, read_config 부분에 order를 불러오는 부분을 추가하였습니다.
기본적인 읽는 방법은 이전 방식과 똑같이 s와 p 포인터를 이용하였습니다.
만약 check_valide_order함수를 통해서 유효성 검증을 실패했을 경우, 에러창을 띄우며 config파일의 다음 라인으로 넘어가게끔 구현하였습니다.
또한 만약 공백일 경우 task.order에 rand() % 100 + 9999를 입력시킵니다.
이를 통해 공백이 아닌 order를 가진 task보다는 무조건 뒤에 실행되지만,
공백 order를 가진 task끼리는 임의의 순서를 가질 수 있도록 하였습니다.
공백도 아니고, 유효성도 검증받은 order 문자열은 atoi 함수를 통해 문자열을 숫자로 바뀌어 task.order에 저장됩니다.

```
static void
spawn_tasks(void)
{
    Task *task;
    //==================================Edit Part=================================================
    for (task = tasks; task != NULL && running; task = task->next)
    {
        spawn_task(task);
        usleep(1000000);//CPU의 멀티 프로세싱때문인지, 분명히 실행 순서는 링크드리스트의
                        //Order순인데도 불구하고, 자꾸 순서가 뒤죽박죽으로 출력되어
                        //usleep(1000000)을 통해서 spawn_task간의 딜레이를 주었더니 정상적으로 출력되었습니다.
    }
    //============================================================================================
}
```

분명히 order순으로 정렬된 링크드리스트를 확인했음에도, task 실행 순서가 엉망으로 출력되어, 혹시 멀티 프로세싱 때문에 task 실행 순서와 다르게 출력되는게 아닐까? 라는 의심을 하여 spawn_tasks에 usleep문장을 추가하였습니다.
usleep문장을 통해 spawn_task간의 딜레이를 주었더니 정상적으로 order순으로

실행되는 것이 출력되었습니다.

시현에 사용한 config1.txt 파일을 비교하며 프로그램 실행 모습을 보여드리겠습니다.

```
# normal tasks
notcmd:once:101::./TTT
id1:once:1::./task -n Task1 -w Hi -r -t 1
id2:once:5:id1:./task -n Task2 -w Hello -r -t 1
idwait1:wait:28::./task -n Task3 -t 2
respawn1 : respawn :8: : ./task -n Task4 -t 4
idonce1:once:101::./task -n Task5 -t 1
idonce2:once:153::./task -n Task6 -t 2
```

위의 config파일에서

order순서대로 진행된다면

order값이 1인 id1이 실행된 후

order값이 5인 id2가 실행된 다음

order값이 8인 respawn1이 실행되고,

order값이 28이지만 action이 wait인 idwait는 예외처리로 오류가 발생할것이고,

order값이 101이지만 먼저 나온 notcmd는 명세사항상 보다 위 행에 위치하므로 먼저 실행되어야하며

order값이 101이지만 보다 아래 행에 존재하는 idonce1은 그 다음 실행되며

order값이 153인 idonce2가 마지막으로 실행되어야 합니다.

즉 실행 순서는

Task1->Task2->Task4->./TTT->Task5->Task6의 순서로 실행되어야 명세서의 조건을 만족시킵니다.

이 config파일을 이용하여 procman파일을 실행시켜보겠습니다.

```
root@server:~/assignment1-sample# ./procman config1.txt
invalid action 'wait' in line 9, ignored
```

우선 Order값이 28이지만 action이 wait인 idwait는 오류 메시지가 출력되며 예외처리가 정상적으로 완료되었고,

```
'Task1' start (timeout 1)
'Task2' start (timeout 1)
'Task1' receive 'Hello from Task2'
'Task2' receive 'Hi from Task1'
'Task1' end
'Task2' end
'Task4' start (timeout 4)
failed to execute command './TTT': No such file or dire
ctory
'Task5' start (timeout 1)
'Task5' end
'Task6' start (timeout 2)
Got SIGCHLD
'Task4' end
```

위에서 생각했던대로 Task1->Task2->Task4->./TTT->Task5->Task6의 순서로 프로그램이 실행된 것을 확인 할 수 있었습니다.

또한 Order를 4자리 이상인 숫자로 넣었을 경우나, 문자가 들어갔을 경우에 예외처리 또한 잘 처리되는지 확인해보겠습니다.

```
# invalid order
id3:once:10000::./task -n Task1 -w Hi -r -t 1
id4:once:abc::./task -n Task1 -w Hi -r -t 1
```

config파일에 추가한 예외부분입니다.

실행결과는 다음과 같습니다.

```
root@server:~/os# ./procman config1.txt
invalid action 'wait' in line 9, ignored
invalid format in line 15, ignored
invalid id 'id_4' in line 18, ignored
invalid id 'ID3' in line 19, ignored
invalid action 'restart' in line 22, ignored
invalid order '10000' in line 25, ignored
invalid order 'abc' in line 26, ignored
```

맨 아래줄에 Invalid order 10000과 'abc'를 확인할 수 있습니다.

즉 정상적으로 예외처리가 되는 것을 확인할 수 있습니다.

자세한 프로그램 결과는 별첨한 result1.txt에 기록되어 있습니다.

감사합니다!