운영체제(공통반)

리눅스 운영체제에서 프로세스 실행/ 종료를
자동으로 관리하는 프로그램의 추가 기능을 구현.

과목명 : 운영체제(공통반)
교수명 : 양승민 교수
학과 : 산업정보시스템공학과
학번 : 20131847
이름 : 박호정
제출일자 : 2019.10.04

목차

1.소개

이번 과제는 procman 프로그램이 txt 파일을 행 단위로 모두 읽어서 실행할 task의 정보들을 저장한다. 등록된 task들을 실행하고 종료해주는 관리 프로그램이다. 자식 프로세스가 종료되면서 부모 프로세스에 시그널을 보내게 된다. 이 시그널이 처리되는 방식을 변경하기 위해서 sigaction함수를 사용한다. 이번 과제에서는 sigaction함수 대신 signalfd함수로 변경하여 구현하는 과정을 거쳤다.

과제 상황에서는 실행해야할 자식 프로세스가 여러 개 있는데 order라는 우선 순위 정보를 이용하여 먼저 처리될 task를 task list의 앞쪽에 위치시키도록 하였다.

2.관련 연구
2.1 소스 분석 procman.c

procman.c 는 main 함수 위에 모든 변수들과 함수들이 미리 정의되어 있다.
함수들은 static의 storage_class_specifier을 가지도록 구현되어 있었으며 직관적으로 작성되어 있어 코드의 이해가 쉽다.

procman.c를 비롯하여 코드 설명은 postion 순서대로 설명하도록 한다.

-enum Action
프로세스의 once, respawn의 정보를 저장하기 위해 사용.

-struct _Task (=Task)
한 행에서 읽어들인 정보를 저장한다.
id, action, order, pipe-id, command가 각각의 형식에 맞추어 저장되도록 정의되어 있다.

-Task* tasks
읽어들인 정보를 바탕으로 task를 linked_list 형태로 관리한다. 첫 task를 가리키며 이를 통해 task를 접근한다.

-int running
main 함수 실행시 1로 값이 할당되고 terminate_children 함수에서 0으로 할당된다.

-static char *strstrip(char *str)
str의 양쪽 부분에 있는 space를 제거해주는 함수이다.

- static int check_valid_id(const char *str)
id 영역에 입력된 값이 영문 소문자와 2개 이상 8개 이하로 구성되어 있는지 확인한다.
정상적인 경우 0을 리턴하고 아닌경우 -1을 리턴한다.

- static Task * lookup_task(const char *id)
id를 통해 task를 검색하는 함수이다. 존재하면 해당 task의 포인터를 아닌 경우 NULL을 리턴한다.

- static Task * loopup_task_by_pid(pid_t pid)
lookup_task와 동일한 기능이며 pid를 기준으로 검색한다.

-static void print_tasks()

과제 수행 중 구현한 함수로 tasks을 확인하고자 구현하였다. tasks를 통해 모든 task들을 출력한다. 이를 통해 order 기준에 맞추어 task가 등록되어 있는지 확인할 수 있다.

- static void append_task(Task *task)
task를 추가하는 함수이다. 기존 tasks들을 기준으로 들어갈 곳을 찾아서 추가된다. 비교 기준은 order이며 order가 낮은 것이 우선순위가 높다. order 기능을 위해 코드를 변경하였으며 singly linked list 형태로 구현하였다.

-static void print_sp(char *ele, char *s, char *p)
한 행을 파싱할 때 s, p로 파싱되는 부분을 체크하기 위해서 구현해 보았다. debugging 과정에서 사용하였으며 id, action, order 등 정확하게 파싱되는지 체크할 때 이용하였다.

-static int read_config(const char *filename)
config 파일을 입력으로 주면 모든 행에 대하여 반복 수행한다. parsing pattern이 어느정도 정해져 있었다. strchr() 함수를 이용하여 : 단위로 파싱을 진행해 나가는 것을 파악하였다. 이것을 파악하여 order 요소를 파싱할 때 이용하였다.

id, action, oreder, pipe-id, command 총 5개의 요소를 파싱해준다. 한 행별로 while문 한 번이 수행된다. 규격에 맞는 행이 들어왔을 때, 파싱된 요소들을 이용하여 task를 추가해준다. read_config에서 strchr() 함수를 기준으로 p를 :부분에 위치시키고 '\0'으로 값을 변경하여 파싱하는 방법이 핵심이다. %s 가 '\0' 전까지 문자열로 인식하는 것을 응용한 것으로 파악된다.

-static char ** make_command_argv(const char *str)
command로 들어온 것을 2차원 argv 형태로 만들어 리턴해준다. 이 argv는 exec의 인자로 추후에 사용된다.

- static void spawn_task(Task *task)
task를 인자로 받아 프로세스를 실행시킨다. fork-> exec의 전형적인 방법으로 수행된다.

-static void spawn_tasks(void)
등록된 모든 task들을 차례대로 spawn 해준다.

-static void wait_for_children(int signo)
waitpid 함수로 부모프로세스에서 자식프로세스가 실행될 때 BLOCK되지 않도록 세번째 인수인 option을 WNOHANG으로 설정한다. 이후 task의 멤버 action 값이 ACTION_RESPAWN인 경우는 다시 spawn_task로 task를 생성한다.

-static void terminate_children(int signo)
running 변수를 0으로 할당하여 현재 실행되는 프로세스를 끝낸다. 어떤 SIGNAL에 의해 끝났는지 알려준다.

-int main(int argc, char **argv)
: main 함수의 주요 부분은 다음과 같다

1) 변수선언
int 형 변수terminated 선언하여 정상종료된 것이 있는지 확인
sigset_t형 변수 mask로 signal을 저장할 집합 생성
int형 변수 sfd로 signalfd() 함수 리턴값 저장
int 형 변수 signo 선언하여 발생된 signal 저장

2) config-file check

3) config.txt load check

4) 전역 변수 pid_number의 초기값을 getpid()로 설정

5) sigemptyset() 함수로 mask 집합 값 비움

6) sigaddset() 함수로 SIGCHLD를 mask 집합에 더함

7) sigprocmask(SIG_BLOCK, &mask, NULL) 함수 호출로 mask에 넣은 SIGCHLD signal은 사용자에 의해 처리 되도록 함

8) sfd 에 signalfd(-1, &mask, 0) 결과 저장

9) running 값 1로 초기화 하고 spawn_tasks() 실행

9) 구조체 signalfd_siginfo sfd_info 를 생성하고, 시그널을
저장하는 sfd를 검사하면서 signal이 SIGCHLD일 경우 wait_for_children 수행.
위 과정을 무한 반복

2.1.2 소스 분석 task.c

: task.c 는 읽기, 쓰기, 메시지 출력과 같은 연산을 수행. 옵션 별 기능은 다음과 같다.

- n : 프로세스에 "Task"라는 문자열을 붙여줌.

- t : timeout 값에 대한 정보 (optarg)

- r : 읽기 연산 read_stdin 값 1로 설정

- w: 쓰기 연산 msg_stdout 이 optarg를 가리키도록 함
이외의 문자에 대한 연산은 따로 하지 않는다.
이후 task.c의 흐름은 다음과 같다.

1) 구조체 sigaction sa를 선언하여 구조체내의 멤버 sa.sa_mask에 SIGINT, SIGTERM추가하여 해당 signal에 대한 대한 signal 핸들러 등록

2) w인 경우, msg_stdout이 NULL이 아닌 경우는 적힌 메시지를 쓰고, MSG로 해당 프로세스가 메시지를 보냈음을 출력

3) r인 경우, read_stdin 값이 1인 경우는 메시지 읽고 MSG로 해당 프로세스가 메시지를 받았음을 출력

4) while로 timeout이 끝날 때까지 timeout에 주어진 값 후위감소연산 진행

2.3 사용한 시스템 콜, API 함수 설명

2.3.1 pipe()
int pipe (int filedes[2]);
에러발생시 -1 , 성공시 0
: IPC에서 사용하는 파이프 생성 명령어. pipe()에서 생성하는 파이프는 커널에 생성됨.
프로세스에서는 pipe에 대한 file descriptor를 이용하여 사용가능. 이때, filedes[0]은 read-only, filedes[1]은 write-only

2.3.2 fclose()
int fclose (FILE *stream)
성공시 0 return, 실패시 -1 return
개방된 스트림을 닫는다. 버퍼에 남은 출력용 데이터는 파일로 기록, 입력용 데이터는 지워짐(버퍼 비워짐)
에러발생시 NULL 반환

2.3.3 fork()
pid_t fork ();
실패시 -1 return,
성공시 부모에게는 생성된 자식 프로세스의 pid가, 자식 프로세스에는 0이 return.

현재 실행되는 프로세스에 대해 복사본 프로세스 생성. 거의 대부분의 자원을 물려받는다.

2.3.4 execvp()
int execvp (const char* path, char* const argv[])
PATH에 등록된 디렉토리에 있는 프로그램 실행.
다른 프로그램 실행후 자신은 종료함.
path : 실행할 프로그램의 경로
argv : 프로그램 실행시 넘겨줄 인자 목록

2.3.5 usleep()
void usleep(unsigned long useconds)
설정한 us 동안 wait state가 됨.

2.3.6 kill()
int kill (pid_t pid, int sig)
성공시 0, 실패시 -1 return
해당 pid에 sig 전송하는 함수.
pid 의 종류에 따른 의미
- pid > 0 : sig를 pid에 보냄.
- pdi = 0 : 현재 프로세스가 속한 그룹의 프로세스 모두에게 보냄
- pid = -1 : 1번 프로세스 제외하고 다 보냄
- pid < -1 : -pid 프로세스가 포함된 모든 그룹의 프로세스에게 보냄.

2.3.7 memmove()
void memmove (void *dest, const void *src, size_t n);
src번지에 있는 데이터를 dest가 지정하는 번지로 n 바이트만큼 복사

2.3.8 calloc()
void* calloc (size_t nitems, size_t size);
동적으로 할당하되, nitems * size 만큼 메모리를 heap 영역에 할당하면서 해당 영역을 0으로 초기화 한 뒤 반환.

2.3.9 memset()
void memset (void *s, int c, size_t n);
* s : 설정할 메모리 시작 주소
* c : 8bit 짜리 값
* n : 개수
* s가 가리키는 메모리를 값 c로 n개 채움

2.3.10 sigaction()
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)
시그널 처리를 결정하는 함수
* int signum : 시그널 번호
* struct sigaction *act : 새롭게 지정할 처리 행동
* struct sigaction *oldact : 이전행동,
* 시그널 처리 성공시 0반환, 실패시 -1 반환

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

2.3.11 sigemptyset()
int sigemptyset(sigset_t *set)
시그널 여러개를 하나의 집합으로 묶어줄 때에 빈 집합을 만들어주는 함수.
집합 변수를 성공적으로 비울 시 0반환, 실패시 -1반환
* sigset_t* set : 시그널 집합 변수

2.3.12 waitpid()
pid_t waitpid (pid_t pid, int* status, int options);
pid_t pid : 감시 대상인 자식 프로세스의 ID.
(다른 값 지정 가능)

-1: 여러자식중 하나라도 종료되면 복귀
0:같은 그룹의 자식 프로세스 종료시 복귀
양수: pid에 해당하는 자식 프로세스 종료시 복귀)
* int* status : 자식 프로세스의 종료 상태 정보
* int options : 대기(wait)를 위한 옵션
* (WNOHANG : 자식 프로세스가 종료되었는지 실행중인지만 확인하고 바로 복귀.
부모 프로세스가 block되지 않음
* 0 : 자식 프로세스가 종료될때까지 부모 프로세스 block)
반환값은 정상일경우 종료된 자식 pid 반환, 실패시 -1 반환,
WNOHANG 사용하고 자식프로세스 종료되지 않았을 경우 0 반환

2.3.13 dup2()
* int dup2 (int fildes, int fildes2); 파일 디스크립터 복사본 만듬.
dup()은 커널에서 사용하지 않는 디스크립터 번호중 하나를 자동으로 생성하나, dup2()는 프로그래머가 원하는 번호로 지정할 수 있음.
단, dup2()에서 프로그래머가 지정한 번호가 이미 사용하는 번호라면 dup2()는 자동으로 그 파일을 닫고 다시 지정해줌.
성공시 원하는 파일 디스크립터 번호 반환, 실패시 -1 반환

2.4 makefile

makefile은 컴파일과 링킹 자동화를 위한 도구이다.

세 개의 c 파일을 합쳐서 실행파일을 만들 경우
% gcc -c main.c
% gcc -c foo.c
% gcc -c fun.c

각각 object 파일을 만든 후 링킹을 시켜 실행파일을 만들어야 한다.
또한, clean이나 각종 환경변수 설정을 하게 되는 경우 반복되는 작업들이 많이 발생한다.
따라서, makefile을 통하여 반복되는 작업을 재사용하는 효과를 얻을 수 있다.

object 파일이란 c파일 자체를 컴파일하여 얻은 binary 파일이다. main.c -> main.o , foo.c -> foo.o 가 된다.
여러 object 파일이 있다면 각각의 object 파일(binary, 기계어 파일)을 우선 만들고 이들을 링킹하여 실행파일을 만든다.

make 파일의 구조는 다음과 같다.

1.목표파일 이름 : 2. 목표파일을 만드는데 필요한 구성요소 이름들
(탭)3.명령어 목록들

makefile 내에서 variable assignment 방법
:= 는 해당되는 라인에서 할당되는 방식 (c언어에서 assignment 방식과 유사, 즉시 그 라인에서 사용 가능한 것들로 구성됨.)
= 는 compiler bnf 방식과 유사
ex)
example of :=
A := 3
B := $(A) --> A가 미리 선언되어 있어야함.

example of =
C = $(D) -> 아래에 recursive 하게 정의되있을 것으로 예상함.
D = HELLO + $(E) -> HELLO는 그냥 사용하고 E는 아래에 나올 것 예상.
E = WORLD

how to use variable
$(variable name) 으로 사용한다.
ex)
NUM = 10
$(NUM)

how to append more text to variable
+= word 하면 ' word'이 추가됨.
ex)
object = main.o foo.o (object assign 'main.o foo.o')
object += fun.o (object change to 'main.o foo.o fun.o' with single space front of 'fun.o')

object is 'main.o foo.o fun.o'

사용된 예약어 설명
$@ : target
$< : first dependency
$^ : all dependency

ex)
all : library.cpp main.cpp
x : y -> x evaluates to y
$@ : all
$< : library.cpp
$^ : library.cpp main.cpp

.PHONY 사용 이유
makefile 이 위치한 폴더에 target name과 중복되는 파일이 이미 있을 수도 있기 때문에 사용한다.
make clean 같은 경우 clean이라는 파일명이 이미 파일로 존재할 수도 있다.
이 때 .PHONY : clean 을 미리 해두면 이미 존재하는 clean파일과 상관 없이 make clean을 이용할 수 있다.
clean 명령어를 실행해도 clean 파일이 별도로 생성되지 않는다. 따라서 이런 키워드 들에 대해서는 .PHONY에 등록해두는 것이 좋다.

symbol '%'
% means string like 'main'
ex)
%.o : %.c
(tab) $ gcc -c $^ -o $@
%에 main이 온다면 main.o : main.c로 사용된다.

3.추가 기능 구현 방법

3.1 시그널

```
int main (int argc, char **argv)
{
  struct sigaction sa;
  int terminated;
  int sfd;
  unsigned int signo;
  sigset_t mask;

  // config.txt file checking
  if (argc <= 1){
      MSG ("usage: %s config-file\n", argv[0]);
      return -1;
    }

  // config.txt load check
  if (read_config (argv[1])){
      MSG ("failed to load config file '%s': %s\n", argv[1], STRERROR);
      return -1;
    }

  pid_for_non_order = getpid();
  sigemptyset(&mask);
  sigaddset(&mask, SIGCHLD);

  // block 할 signal들 세팅
  if(sigprocmask(SIG_BLOCK, &mask, NULL) < 0 ){
    perror("sigprocmask() error\n\n\n");
    return -1;
  }

  // call signalfd
  sfd = signalfd(-1, &mask, 0);
  if(sfd < 0){
    perror("signalfd() error\n\n\n");
    return -1;
  }

  running = 1;
  spawn_tasks ();

  while(1){
    struct signalfd_siginfo sfd_info;
    read(sfd, &sfd_info, sizeof(sfd_info));

    signo = sfd_info.ssi_signo;
    if(signo == SIGCHLD)
    {
      wait_for_children(signo);
    }
```

시그널 구현 방법 설명

- sfd 변수 선언하여 descriptor 저장
- signo 변수 선언하여 signal 값 저장
- mask 변수 선언
- 기존의 argv, argv[1] check code
- read_config로 파일 read
- pid assign 하여 order 정보가 없는 것들 처리.
- signalmask(SIG_BLOCK, &mask, NULL) 함수 호출하여 블록될 mask 지정
- 파일 다 읽은 후 spawn task
- 구조체 signalfd_siginfo sfd_info 선언
- 파일 디스크립터 읽으면서 시그널 체크
- SIGCHLD 인 경우 wait_for_children 수행

3.2 order 기능

parsing pattern을 이용하여 입력값이 규칙에 맞는 order 정보인지 확인하는 부분이다. order 정보가 없는 경우 임의로 지정하고 정확하게 들어온 경우 4자리 이내의 숫자를 order로 지정한다. 이외의 경우 예외 처리를 하였다.

다음은 parsing 하는 부분이다.

```
    /* order */
    s = p + 1;
    p = strchr(s, ':');
    print_sp("order",s,p);
    if(!p)
      goto invalid_line;
    *p = '\0';
    strstrip(s);
    print_sp("order",s,p);

//      if(s[0]=='\0')
//        MSG("s is NULL");

    if(s[0] == '\0'){
      task.order = pid_for_non_order++;
    }else if(atoi(s)){
  if(strlen(s) >4){
        MSG ("invalid order '%s' in line %d, ignored\n", s, line_nr);
        continue;
      }
      task.order = atoi(s);
    }else{
      MSG ("invalid order '%s' in line %d, ignored\n", s, line_nr);
      continue;
    }
```

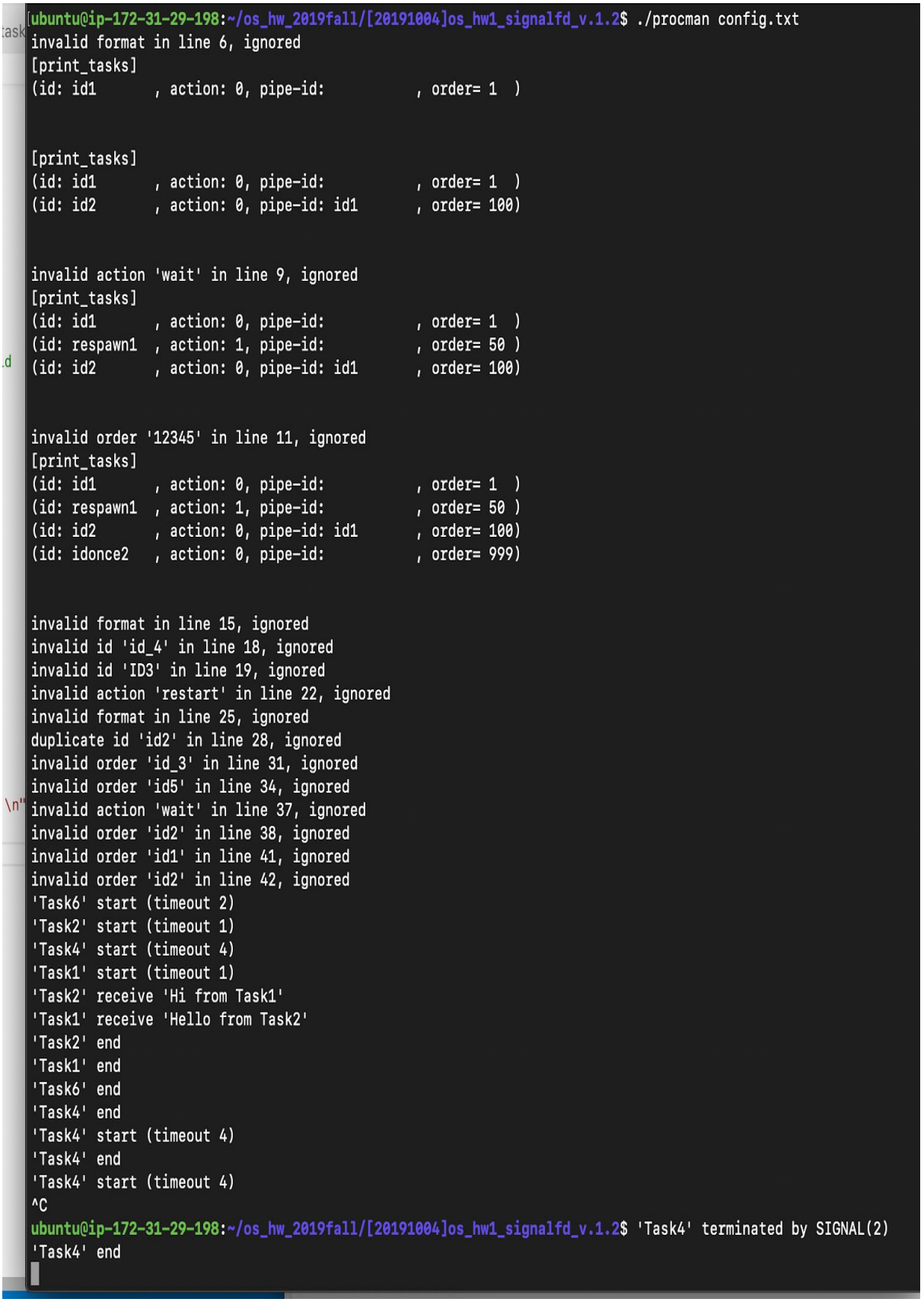
다음은 task 정보를 order를 기준으로 append하는 함수이다. linked list를 이용하였으며 order 정보를 기준으로 list를 탐색하고 맞는 위치에 맞추어 추가해준다.

```
static void append_task (Task *task) // append task by order
{
  Task * new_task;
  new_task = malloc (sizeof (Task)); // task 저장할 메모리 할당
  if (!new_task)
    {
      MSG ("failed to allocate a task: %s\n", STRERROR);
      return;
    }

  *new_task = *task; // new task에 task 내용을 저장
  new_task->next = NULL; // next는 우선 없도록 저장

  if (tasks == NULL) // task가 하나도 없는 경우
    tasks = new_task;
  else
    {
      Task *tp;
      Task * prev_tp;
      tp = tasks;
      prev_tp = tp;
      while(tp!= NULL){

        if(tp->order <= new_task->order){
          prev_tp = tp;
          tp = tp->next;
        }else{
          break;
        }
      }
        MSG("prev_tp:%s\n", prev_tp->id);
        if (prev_tp->next != NULL)
            MSG("prev_tp->next:%s\n", prev_tp->next->id);

       // insert task by non descent order
      new_task->next = prev_tp->next;
      prev_tp ->next = new_task;
    }

  print_tasks();
}
```

다음은 tasks 의 연결이 정확하게 되어있는지 확인하는 부분이다. 모든 task를 순서대로 출력하여 order대로 되어있는지 확인위해 사용하였다.

```
static void print_tasks() // print tasks for check order
{
  MSG("[%s]\n",__func__ );

  Task * tp = tasks;
  // print tasks after tp
  while(1){
     MSG("(id: %-10s, action: %d, pipe-id: %-10s, order= %-3d)\n", tp->id, tp->action, tp->pipe_id, tp->order);
    if(tp->next == NULL){
      break;
    }
    else{
      tp = tp->next;
    }
  }
  MSG("\n\n");
}
```

실행화면


ubuntu@ip-172-31-29-198:~/os_hw_2019fall/[20191004]os_hw1_signalfd_v.1.2$ ./procman config.txt
invalid format in line 6, ignored
[print_tasks]
(id: id1 , action: 0, pipe-id: , order= 1)
[print_tasks]
(id: id1 , action: 0, pipe-id: , order= 1)
(id: id2 , action: 0, pipe-id: id1 , order= 100)
invalid action 'wait' in line 9, ignored
[print_tasks]
(id: id1 , action: 0, pipe-id: , order= 1)
(id: respawn1 , action: 1, pipe-id: , order= 50)
(id: id2 , action: 0, pipe-id: id1 , order= 100)
invalid order '12345' in line 11, ignored
[print_tasks]
(id: id1 , action: 0, pipe-id: , order= 1)
(id: respawn1 , action: 1, pipe-id: , order= 50)
(id: id2 , action: 0, pipe-id: id1 , order= 100)
(id: idonce2 , action: 0, pipe-id: , order= 999)
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 format in line 25, ignored
duplicate id 'id2' in line 28, ignored
invalid order 'id_3' in line 31, ignored
invalid order 'id5' in line 34, ignored
invalid action 'wait' in line 37, ignored
invalid order 'id2' in line 38, ignored
invalid order 'id1' in line 41, ignored
invalid order 'id2' in line 42, ignored
'Task6' start (timeout 2)
'Task2' start (timeout 1)
'Task4' start (timeout 4)
'Task1' start (timeout 1)
'Task2' receive 'Hi from Task1'
'Task1' receive 'Hello from Task2'
'Task2' end
'Task1' end
'Task6' end
'Task4' end
'Task4' start (timeout 4)
'Task4' end
'Task4' start (timeout 4)
^C
ubuntu@ip-172-31-29-198:~/os_hw_2019fall/[20191004]os_hw1_signalfd_v.1.2$ 'Task4' terminated by SIGNAL(2)
'Task4' end